CodeWarrior™ Development Studio for StarCore® DSP Architectures

v2.6

SC100 Assembler User Guide

Revised 24 September 2004

metrowerks

Metrowerks and the Metrowerks logo are registered trademarks of Metrowerks Corporation in the United States and/or other countries. CodeWarrior is a trademark or registered trademark of Metrowerks Corporation in the United States and/or other countries. All other trade names and trademarks are the property of their respective owners.

Copyright © 2003—2004 Metrowerks Corporation. ALL RIGHTS RESERVED.

No portion of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Metrowerks. Use of this document and related materials are governed by the license agreement that accompanied the product to which this manual pertains. This document may be printed for non-commercial personal use only in accordance with the aforementioned license agreement. If you do not have a copy of the license agreement, contact your Metrowerks representative or call 1-800-377-5416 (if outside the U.S., call +1-512-996-5300).

Metrowerks reserves the right to make changes to any product described or referred to in this document without further notice. Metrowerks makes no warranty, representation or guarantee regarding the merchantability or fitness of its products for any particular purpose, nor does Metrowerks assume any liability arising out of the application or use of any product described herein and specifically disclaims any and all liability. **Metrowerks software is not authorized for and has not been designed, tested, manufactured, or intended for use in developing applications where the failure, malfunction, or any inaccuracy of the application carries a risk of death, serious bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft navigation or communication, emergency systems, or other applications with a similar degree of potential hazard**.

How to Contact Metrowerks

Corporate Headquarters	Metrowerks Corporation 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	`http://www.metrowerks.com`
Sales	United States Voice: 800-377-5416 United States Fax: 512-996-4910 International Voice: +1-512-996-5300 E-mail: `sales@metrowerks.com`
Technical Support	United States Voice: 800-377-5416 International Voice: +1-512-996-5300 E-mail: `support@metrowerks.com`

Contents

1 Introduction 9

Assembler 9
Software Development Flow. 9
References. 11

2 SC100 Assembler 13

Starting the Assembler 13
Command-Line Options 14
Using an Environment Variable 17
Reading Input from an Argument File 17
Generating an Object File 18
Modifying Object File Contents 18
Redirecting the Source Listing 19
Controlling Assembler Messages 20
Searching Additional Directories 20
Defining Substitution Strings 21
Using OPT Options on the Command Line 21
Specifying a Target Architecture. 21
Specifying Endian Mode. 22
Checking Programming Rules 23
Data Analysis 28
Assembler Processing. 35
Source Statements. 36
Label Field. 37
Operation Field 37
Operand Field 38
Comment Field 38
Variable Length Execution Sets 38
Symbol Names 39
Symbol Labels. 39
Strings 40
Source Listing 40

Contents

Source Listing Example. 41
Final Information 43

3 Expressions 45

Absolute and Relative Expressions 45
Expression Memory Space Attributes 45
Internal Expression Representation. 46
Constants 46
Numeric Constants 46
String Constants. 47
Operators 48
Operator Precedence 50
Functions 50
ABS Absolute Value. 53
ACS Arc Cosine 53
ARG Macro Argument 54
ASN Arc Sine 54
AT2 Arc Tangent. 55
ATN Arc Tangent 55
BIGENDIAN Endian Mode Check 55
CCC Cumulative Cycle Count 56
CEL Ceiling 56
CHK Instruction/Data Checksum. 57
CNT Macro Argument Count. 57
COH Hyperbolic Cosine. 57
COS Cosine 58
CTR Location Counter Number 58
CVF Convert Integer to Floating Point. 59
CVI Convert Floating Point to Integer 59
CVS Convert Memory Space 59
DEF Defined Symbol 60
EXP Expression Check. 60
FLD Shift and Mask 61
FLR Floor 62
FRC Convert Floating Point to Fractional 62

For More Information: www.freescale.com

INT Integer Check 62
L10 Log Base 10 63
LCV Location Counter Value 63
LEN String Length 64
LFR Convert Floating Point to Long Fractional 64
LNG Concatenate to Double Word 65
LOG Natural Logarithm 65
LST LIST Directive Flag Value 66
LUN Convert Long Fractional to Floating Point 66
MAC Macro Definition 66
MAX Maximum Value 67
MIN Minimum Value 67
MSP Memory Space 67
MXP Macro Expansion 68
POS Position of Substring 68
POW Raise to a Power 69
REL Relative Mode 69
RND Random Value 69
RVB Reverse Bits in Field 70
SCP Compare Strings 70
SGN Return Sign 70
SIN Sine 71
SNH Hyperbolic Sine 71
SQT Square Root 72
TAN Tangent 72
TNH Hyperbolic Tangent 72
UNF Convert Fractional to Floating Point 73
XPN Exponential Function 73

4 Software Project Management 75

Sections 75
Section Names 75
Nested and Fragmented Sections 76
Sections and Symbols 77
Macros and DEFINE Symbols within Sections 78

Sections and Relocation 78
Address Assignment 78
Overlays 79
Overlay Manager 80
Overlay Example 81
Multi-Programmer Environment Example 84
Method 1: Absolute Mode 85
Method 2: Relative Mode 86

5 Assembler Directives 87

Significant Characters 87
Directive List 88
Descriptions 91
; Start Comment 92
;; Start Unreported Comment 92
\ Continue Line 93
\ Concatenate Macro Argument 93
? Substitute Macro Value 94
% Substitute Macro Hex Value 95
^ Override Macro Local Label 96
" Delimit Macro String 96
" Expand DEFINE Quoted String 97
@ Start Function 97
* Substitute Location Counter 98
++ Concatenate Strings 98
[ ] Delimit Substring 98
[ ] Group Instructions 99
< Force Short Addressing 99
> Force Long Addressing 100
Use Immediate Addressing 100
#< Force Immediate Short Addressing 101
#> Force Immediate Long Addressing 101
ALIGN Align Location Counter 102
BADDR Set Buffer Address 102
BSB Allocate Bit-Reverse Buffer 103

BSC Allocate Constant Storage Block 104
BSM Allocate Block Storage for Modulo 105
BUFFER Start Buffer 106
COMMENT Start Comment Lines 107
DC Define Constant 108
DCB Define Constant Byte 109
DCL Define Constant Long 110
DEFINE Define Substitution String 110
DS Define Storage 111
DSM Define Storage for Modulo 112
DSR Define Reverse-Carry Storage 113
DUP Duplicate Source Lines 114
DUPA Duplicate Sequence with Arguments 115
DUPC Duplicate Sequence with Characters 116
DUPF Duplicate Sequence in Loop 117
ELSE Start Alternative Conditional Assembly 118
END End of Source Program 118
ENDBUF End Buffer 119
ENDIF End Conditional Assembly 119
ENDM End Macro Definition 120
ENDSEC End Section 121
EQU Equate Symbol to Value 121
EXITM Exit Macro 122
FAIL Issue Programmer Error Message 122
FALIGN Align with Fetch-Set 123
GLOBAL Declare Global Section Symbol 124
GSET Set Global Symbol to Value 125
HIMEM Set High Memory Bounds 125
IF Start Conditional Assembly 126
INCLUDE Include Secondary File 127
LIST List Assembly 128
LOMEM Set Low Memory Bounds 129
MACLIB Specify Macro Library 129
MACRO Define Macro 130
MSG Issue Programmer Message 131

NOLIST Stop Assembly Listing 132
NOTE Include Note 132
OPT Set Assembler Options 133
ORG Initialize Memory Space and Location Counters 139
PAGE Advance Page or Size Page 140
PMACRO Purge Macro Definition 141
PRCTL Send Control String to Printer 142
RADIX Set Constant Radix 143
SECFLAGS Set ELF Section Flags 143
SECTION Start Section 144
SECTYPE Set ELF Section Type 147
SET Set Symbol to Value 148
SIZE Set Symbol Size 149
STITLE Initialize Program Subtitle 149
TITLE Initialize Program Title 150
TYPE Set Symbol Type 151
UNDEF Undefine DEFINE Symbol 151
WARN Issue Programmer Warning 152
XDEF Define External Symbol 153
XREF Reference External Symbol 154

6 Macros and Conditional Assembly 155

Defining a Macro 155
Calling a Macro 156
Macro Expansions 157
Macro Libraries 157
Dummy Argument Operators 157
Macro Directives 158
Conditional Assembly 158

A ASCII Character Codes 161

B Assembler Messages 163

Index 189

1

Introduction

The SC100 assembly language tools consist of an assembler, a linker, an archiver, and several utilities. This manual explains the assembly language tools for the StarCore SC100 generation of DSP cores: SC110 and SC140.

(For information on the C compiler, see the *SC100 C Compiler User's Manual*. For information on the simulator, see the *SC100 Simulator Reference Manual*.)

Assembler

The SC100 Assembler converts handwritten or compiler-generated SC100 assembly code into ELF object files. The assembler provides:

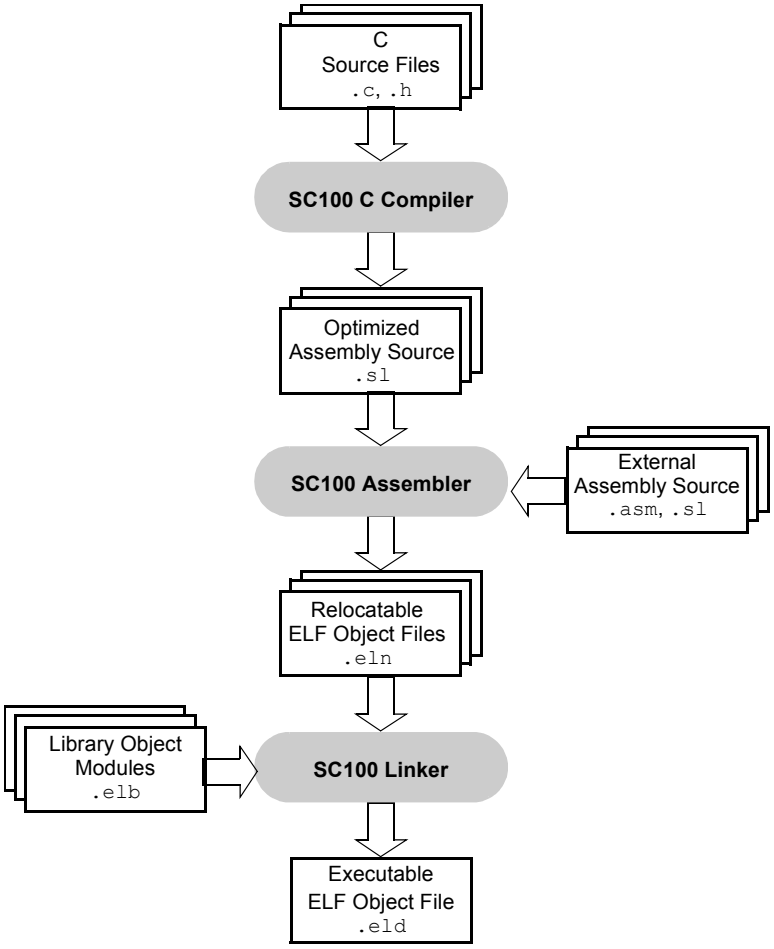
- Expression evaluation involving numeric constants, string constants, operators, and built-in functions
- Modular programming involving sections
- Macros that allow variable arguments
- Conditional assembly

Chapters 2 through 6 provide a complete explanation of the assembler.

Software Development Flow

Figure 1.1 illustrates the software development flow, showing the inputs and outputs of each stage.

Figure 1.1 StarCore Development Tools



The SC100 assembler architecture is compiler-friendly, so you can combine it with a compiler that generates exceptionally compact code. In this manner, you can write applications in C, yet achieve code density and high performance comparable to that of hand-coded assembly programs.

Compiler options let you specify appropriate settings, development tools, processing stages, and processing options. Then, when you submit C source files to the compiler, its shell automatically advances the source files through compilation, assembly, and linking, to produce an executable program.

You submit hand-coded assembly language files to the assembler, which transforms the assembly code into ELF object code. The assembler then submits this object code and any object modules to the linker, which combines them into a single, executable program.

You may load this executable program into the simulator for execution and evaluation.

References

- *SC100 Application Binary Interface* (MNSC100ABI/D)
- *SC100 C Compiler User's Manual* (MNSC100CC/D)
- *SC100 Simulator Reference Manual* (MNSC100SIM/D)
- *SC110 DSP Core Reference Manual* (MNSC110CORE/D)
- *SC140 DSP Core Reference Manual* (MNSC140CORE/D)

2

SC100 Assembler

The SC100 Assembler translates hand-written or compiler-generated SC100 assembly language programs into machine language. The assembler uses executable and linking format (ELF) for object files. It supports these features:

- Expression evaluation using numeric constants, string constants, operators, and built-in functions
- Modular programming using sections
- Macros that allow variable arguments
- Conditional assembly

Starting the Assembler

Use this command to start the assembler:

`asmsc100 [option ...] file ...`

where:

`option`

One or more of the options Table 2.1 lists.

`file`

One or more assembly source files, separated by spaces. The assembler processes these files in the order listed, assembling them into a single object file.

If a file's name does not include an extension, the assembler tries to open the file without an extension. If that is not successful, the assembler appends `.asm` to the name, and again tries to open the file.

If the `asmsc100` command does not have any options, the assembler processes the source file, outputting a source listing to the standard output.

NOTE The assembler does not generate an object file unless the command includes the `-b` option.

This example command starts the assembler, assembles source file `corr.asm`, outputs a source listing to the standard output, and generates relocatable object file `corr.eln`. Except for the `-b` option, this command tells the assembler to use all default settings.

```
asmsc100 -b corr.asm
```

Command-Line Options

Table 2.1 summarizes assembler options; later sections of this chapter explain these option in more detail. Note that:

- Assembler options are not case sensitive.
- Certain options may appear more than once on the command line, as their descriptions explain.
- Arguments can follow their option letter immediately, a space can separate the option letter and an argument, or an equal sign (=) can separate the option letter and an argument. (Option descriptions indicate the exceptions to this rule.)

However, the `-l` and `-b` options are ambiguous if spaces separate them and they immediately precede source file names, as in the command line:

```
asmsc100 -l -b main io
```

The assembler acts as though `main` were a source file, so tries to open `main` for reading. If `main` instead is an argument for the `-b` option, you must clearly identify it by doing one of these actions:

- Deleting the space before `main`: `asmsc100 -l -bmain io`.
- Replacing the space with an equal sign: `asmsc100 -l -b=main io`.
- Repositioning another option before the list of source file names: `asmsc100 -l -b main -v io`.

Table 2.1 SC100 Assembler Options

Option	Description	See Section
`-a`	Specifies absolute mode instead of the default relative mode. With the `-b` option, generates an executable object file.	Generating an Object File
`-arch <arch>`	Specifies the assembly architecture for the file. The arch parameter values include: `sc110`, `sc120`, `sc1200`, `sc140`, `sc140e`, `sc1400`, `msc8101`, `msc8102`, and `pog`. (Follow the same `scxxx` pattern for later architecture variants.) This option is synonymous with the command osc110|sc120|sc1200|sc140|sc140e|sc1400. Another valid parameter value is `rainbow`; with the `-s all` option, it activates specific restriction checking. Still another valid parameter value is `starlite`, but this architecture is not compatible with any `BMTSET.W` instructions in your code.	Specifying a Target Architecture
`-b[`*`objfile`*`]`	Generates an object file, assigning the specified name.	Generating an Object File
`-d`*`symbol string`*	Defines substitution strings to be used on all source lines; equivalent to the `DEFINE` directive.	Defining Substitution Strings
`-ea` *`errfile`*	Appends the standard error output stream to the specified file. A space is mandatory before the file name.	Controlling Assembler Messages
`-ew` *`errfile`*	Writes the standard error output stream to the specified file, overwriting any previous contents. A space is mandatory before the file name.	Controlling Assembler Messages
`-f`*`argfile`*	Reads options and file names from the specified file, appending them to the command line.	
`-g`	Adds debug information to the object file; valid only with the `-b` option.	Modifying Object File Contents

Table 2.1 SC100 Assembler Options (*continued*)

Option	Description	See Section
`-i`*`pathname`*	Adds the specified directory to the standard search paths; repeatable multiple times. The assembler searches directories in their command-line order.	Searching Additional Directories
`-k[conv_file]`	Enables data analysis; `conv_file` is the name of the initialization file.	Data Analysis
`-l[`*`lstfile`*`]`	Redirects the source listing to the specified file.	Redirecting the Source Listing
`-m`*`directory`*	Specifies the directory that contains macro definitions; repeatable multiple times. The assembler searches directories in their command-line order. Equivalent to the `MACLIB` directive.	Searching Additional Directories
`-o`*`opt`*`[,`*`opt`*`...]`	Designates assembler options; commas without spaces must separate multiple options. Equivalent to the `OPT` directive; valid arguments are any `OPT`-directive options.	Using OPT Options on the Command Line
`-q`	Specifies quiet mode; suppresses the assembler banner.	Controlling Assembler Messages
`-s {all|`*`id`*`[,`*`id`*`...]}`	Enables checks of programming rules (formerly called *restriction checking*). Commas without spaces must separate multiple arguments. (This option replaces the `-suppress` option of earlier releases.) With the `-arch rainbow` option, activates specific restriction checking.	Checking Programming Rules
`-u all|none|restr[, restr...]`	Inhibits restriction checking for the specified restrictions, as in `-u a1,a2` or `-u all`. If the command line includes both `-s` and `-u` options, the order does matter regarding assembler behavior.	
-v	Specifies verbose mode, reporting assembly progress.	Controlling Assembler Messages
-z	In COFF mode only, strips symbol information from the executable object file. Valid only with the `-a` and `-b` options. (Has no effect in ELF mode.)	Modifying Object File Contents

Using an Environment Variable

If you use command-line options regularly, you may assign them to the environment variable DSPASMOPT. Before processing any options, the assembler adds this variable's text to the existing command line.

To define DSPASMOPT:

1. In Table 2.2, find the command line and environment file appropriate for your operating system. (The environment file is in the directory that `$SC100_HOME` defines.)
2. Enter the command line in the environment file.
   a. Any option of Table 2.1 is a valid `option` parameter value.
   b. Separate multiple option values with spaces.
   c. Start each option value with a hyphen.
3. When you are done, re-execute the environment file.

Table 2.2 DSPASMOPT Command Lines

Operating System	Command Line	Environment File
UNIX: Bourne shell (sh, ksh, bash)	`DSPASMOPT="-option ..."` `export DSPASMOPT`	`env.sh`
UNIX: C shell (csh, tcsh)	`setenv DSPASMOPT "-option ..."`	`env.csh`
Windows	`set DSPASMOPT=-option ...`	`env.bat`

For example, if the DSPASMOPT definition in the `env.sh` file is:

```
DSPASMOPT="-b -l"
export DSPASMOPT
```

Then each time you invoke the assembler, it adds the `-b` and `-l` options to the command line. The command `asmsc100 corr.asm` becomes `asmsc100 -l -b corr.asm`.

Reading Input from an Argument File

The `-fargfile` option instructs the assembler to read command-line input from the specified argument file. This option is a method for passing command-line input from such an argument file to the assembler.

The `argfile` parameter value can include an optional pathname. You may repeat this option multiple times.

For example, this command invokes the assembler, telling it to read arguments from the file `asmopts`:

```
asmsc100 -fasmopts -q filter.asm
```

An argument file is a text file containing a list of options, arguments, file names — even the `-f` option itself. Within the argument file, a space, blank, tab, or newline character must separate each file or option. Use semicolons to include comments.

Argument-file contents can be as simple as this example:

```
-b -l
-s a1,gg4
```

Generating an Object File

The assembler generates a relocatable object file only if the command line includes the `-b` option. If the command line includes both the `-a` and `-b` options, the assembler instead generates an executable object file.

`-a`

With the `-b` option, generates an executable object file.

`-b[`*`file`*`]`

Generates an object file, assigning the specified name; the file name may include an optional pathname. Using a hyphen in place of a file name sends the object file to the standard output.

This option overwrites any file that has the same name. If this option does not include a name, the assembler uses the name of the first source file in the command line, with extension `.eln` (relocatable file) or `.eld` (executable file).

This example assembles files `main.asm` and `fft.asm` into the executable object file `filter.eld`:

```
asmsc100 -a -bfilter.eld main.asm fft.asm
```

Modifying Object File Contents

To modify the contents of the option file, use the `-g` or `-z` options:

`-g`

Adds these debugging sections to the object file: `.debug_abbrev`, `.debug_aranges`, `.debug_delay_slots`, `.debug_info`, `.debug_macinfo`, `.debug_loc`, and `.debug_line`. Produces debug information for all global symbols, including EQUs. Wherever necessary for debugging, inserts local symbols (named F_MemAllocArea_[section_name]_[pc]).

This option is valid only with the `-b` option.

Only the assembler can generate the `.sc100.delay_slots` section; this section is not part of any assembly file that the compiler produces. Accordingly, for an assembly file from the compiler, your command line should include the `-c` option, to suppress assembly source-level debug information. But for a manually written assembly file, your command line should not include the `-c` option.

If the assembly file includes an overlay or union section, the assembler appends that section's name to the names of debug sections. For consistent debug information, modules that contain debug information and an overlay/union section must not include other text sections.

`-z`

In COFF mode only, strips symbol information from the executable object file. This option is valid only with both the `-a` and `-b` options. (This option has no effect in ELF mode.)

This example strips symbol information from the executable object file `filter.eld`:

```
asmsc100 -a -bfilter.eld -z filter.asm
```

Redirecting the Source Listing

Per the default setting, the assembler sends a source listing to the standard output. To save the source listing to a file, use the `-l` option.

`-l[`*`file`*`]`

Redirects the source listing to the specified file; the file name may include an optional pathname.

This option overwrites any file that has the same name. If this option does not include a name, the assembler uses the name of the first source file in the command line, with extension `.lst`.

This first example assembles files `filter.asm` and `gaus.asm` into the single, relocatable object file `filter.eln`, then redirects the source listing to file `filter.lst`:

```
asmsc100 -b -lfilter.lst filter.asm gaus.asm
```

The second example inhibits the source listing, by specifying `IL` (inhibit listing) as an argument for the `-o` option. (Another way to inhibit the source listing is specifying `IL` as an argument to the OPT directive in the assembly source file.)

```
asmsc100 -b -oil filter.asm gaus.asm
```

Controlling Assembler Messages

To redirect the standard error output stream to a file, and to control the level of messages the assembler displays, use the `-ea`, `-ew`, `-q`, or `-v` options.

`-ea` *file*

Appends the standard error output stream (`stderr`) to the specified file. A space is required between `-ea` and the file name.

`-ew` *file*

Writes the standard error output stream (`stderr`) to the specified file, overwriting the file if it already exists. A space is required between `-ew` and the file name.

`-q`

Specifies quiet mode, suppressing the assembler banner.

`-v`

Specifies verbose mode, reporting assembly progress (such as beginning of passes, and opening and closing of input files) to the standard error output stream. This helps make sure that assembly is proceeding normally.

This example:

- Assembles the files `filter.asm` and `gaus.asm` into the relocatable object file `filter.eln`,
- Redirects the standard error output stream to the file `errors`, and
- Redirects the source listing to the file `filter.lst`.

```
asmsc100 -b -ew errors -lfilter.lst filter.asm gaus.asm
```

Searching Additional Directories

To add directories to the assembler's standard search paths, use the `-i` or `-m` options.

`-i`*pathname*

Adds the specified directory to the search path for INCLUDE files.

`-m`*pathname*

Adds the specified directory to the search path for macro definitions. This option is equivalent to the MACLIB directive.

You may repeat either of these directives multiple times. The assembler searches directories in their command-line order.

These examples add directory `sctools/fftlib` to the search path. (The first example is for a UNIX environment, the second for a Windows environment).

```
asmsc100 -m/sctools/fftlib trans.asm
```

```
asmsc100 -ic:\sctools\fftlib filter.asm
```

Defining Substitution Strings

To define substitution strings, use the `-d` option:

`-dsymbol string`

> Directs the assembler to replace every occurrence of symbol in the source file with the specified string. A space must precede the string. If the string contains spaces, single quotes must enclose the string.
>
> You can repeat the `-d symbol string` sequence multiple times.

This example substitutes the string `1` for all occurrences of `BIG_ENDIAN` in the source file `vit.asm`.

```
asmsc100 -b -dB_END '1' -obe vit.asm
```

Another way to define substitution strings is using the DEFINE directive in the source file.

Using OPT Options on the Command Line

To use any OPT-directive options on the command line, use the `-o` option.

`-oopt[,opt...]`

> Directs the assembler to use the specified OPT-directive options. Commas without spaces must separate multiple options.

This example tells the assembler to include the `MD` and `MEX` options — that is, to include macro definitions and macro expansions in the source listing:

```
asmsc100 -b -l -omd,mex corr.asm
```

Specifying a Target Architecture

There are two ways to specify a target architecture:

- Make it an argument of the `-o` command-line option.
- Make it an argument to the OPT directive, placing it at the beginning of the source file.

The valid arguments are:

- sc110
- sc120
- sc1200
- sc140 *(default)*
- sc140e

- sc1400
- msc8101
- msc8102

Table 2.3 lists the special symbols the assembler automatically defines for an ELF object file. These symbols relate to the architecture and endianness that command line specifies.

Table 2.3 Assembler Symbols

Architecture/ Endianness	Defines
`SC110`	_SC100_, __SC100__, _SC110_, __SC110__
`SC120`	_SC100_, __SC100__, _SC120_, __SC120__
`SC1200`	_SC100_, __SC100__, _SC1200_, __SC1200__
`SC140`	_SC100_, __SC100__, _SC140_, __SC140__
`SC1400`	_SC100_, __SC100__, _SC1400_, __SC1400__
`SC140E`	_SC100_, __SC100__, _SC140_, __SC140__ _SC100_, __SC100__, _SC140E_, __SC140E__
Little	LITTLE_ENDIAN, __LITTLE_ENDIAN__
Big	BIG_ENDIAN, __BIG_ENDIAN__

This first example uses the `-c` command-line option to invoke the assembler for the SC140V1 DSP core:

```
asmsc100 -b -l -osc140e -obe -dBIG_END '1' main.asm fft.asm
vit.asm
```

This second example uses the OPT directive to specify the SC140v1 core architecture:

```
      opt sc140,cex,mex
      page 132,42,0,0,0
LAB1  macro args
      ...
```

Specifying Endian Mode

To specify the endian mode, use the `le` or `be` arguments for the `-o` command-line option or the OPT directive:

`le`

Specifies little-endian object files: the least significant byte occupies the lower word address. This is the default.

`be`

Specifies big-endian object files: the most significant byte occupies the lower word address.

This first example uses the -o command-line option to specify big-endian mode:

```
asmsc100 -b -l -obe vit.asm
```

This second example uses the OPT directive, placed at the beginning of the source file, to specify big-endian mode:

```
      opt  sc140,be
      page 132,42,0,0,0
LAB1  macro args
      ...
```

Checking Programming Rules

The reference manual for each core explains the rules for grouping and sequencing instructions in a variable length execution set (VLES). The assembler enforces static programming rules, marking violations at assembly time. Assembly does not take place if such errors exist.

Table 2.4 lists rule identifiers and the text of associated error messages. Rule identifiers begin actual error messages. For example, the identifier A.1 corresponds to Rule A.1 of the *SC140 Core Reference Manual*.

NOTE The assembler's default setting for rule checking is OFF. But even with this setting, the assembler checks restrictions G.G.1. G.G.2, G.G.3, G.P.1, and L.G.5.
The default setting of previous assembler releases was that rule checking (or restriction checking) was ON; you used the `-suppress` option to disable rule checking. The `-s` option replaces the `-suppress` option.

Use these `-s` option patterns to control rule checking:

`-s` *`id`*`[,`*`id`*`]`

Enables checking for violations of specified rules; `id` values are rule identifiers, without period or other characters. Commas without spaces must separate multiple `id` arguments. Neither the `-s` option nor the `id` arguments are case sensitive. Table 2.4 lists valid `id` values.

`-s all`

Enables checking for all static rules.

`-s none`

Disables checking for all static rules, including G.G.1, G.G.2, G.G.3, G.P.1, and L.G.5.

This first example enables checking for all rules, in source file `myprog.asm`:

```
asmsc100 -b -s all myprog.asm
```

This second example specifies checking for violations of the A.1, A.2, and G.G.1 rules, in source file `myprog.asm`:

```
asmsc100 -b -s a1,a2,gg1 myprog.asm
```

Table 2.4 Programming Rule Error Messages

ID	Error Message
A1	A.1: At least two cycles required between MCTL modification and address pointer usage A.1: At least two VLES required between MCTL modification and address pointer usage A.1: violation of A.1 restriction within short/long loop
A2	A.2: AGU register contents are not available for an additional cycle A.2: AGU register contents may be affected by register write in previous cycle A.2: violation of A.2 restriction within short/long loop (Rule A.2 does not apply if a MOVE-like instruction writes the Bn or Mn used for modulo calculation on Rn. This rule does apply to using Bn-8 representation when writing to Rn.)
A2a	A.2a: AGU register contents may be affected by register write in previous cycle A.2a: violation of A.2a restriction within short/long group (At least one cycle is required between a MOVE-like instruction that writes the Bn or Mn register as a data operand, and a MOVE or AGU arithmetic instruction that uses the same register for modulo calculation.)
A3	A.3: A valid group must follow a group containing JT,JF,BT,BF or TRAP
A4	A.4: Updates an AGU register by an AGU A.4: Updates an AGU register by an AGU arithmetic instruction or an address calculation, with a MOVE-like instruction that uses this register as a source (A DOENn, DOENSHn, CONT, or CONTD that writes the LCN cannot be grouped with a MOVE-like instruction that reads the same register.)
A7	A.7: RTSTK/RTSTKD cannot be grouped in a VLES with a MOVE-like instruction that reads the EMR register
D1	D.1: Illegal instruction in delay slot
D2	D.2: Instruction which uses the SR is not allowed in RTED delay slot

Table 2.4 Programming Rule Error Messages (*continued*)

ID	Error Message
D3	D.3: RTE/D and instruction which changes SR not permitted in same group
D4	D.4: Instruction is not allowed in RTSD/RTED/STKD delay slot
D5	D.5: MOVE-like instructions to SR are not allowed to be grouped with JSR/JSRD/BSR/BSRD
D5a	D.5a: Instruction is not allowed in BSRD/JSRD delay slot
D6	D.6: Instructions in the delay slot of delayed return instructions cannot use or change SP
D8	D.8: A MOVE-like instruction that reads the SR register is not allowed in the delay slot of a CONTD instruction
D9	D.9: Instructions that read the EMR register in the delay slot of a RTED or RTSTK/D are not allowed
D9a	D.9a: Instructions that read the EMR register in parallel with a RTE/D or RTSTK/D are not allowed
GG1	G.G.1: Too many total instructions
GG3	G.G.3: Only one bit mask instruction permitted in a VLES G.G.3: Too many total AGU ops G.G.3: Too many total DALU ops plus BFU ops
GG4	G.G.4: Cannot group an instruction that changes bits in EMR with a MOVE-like instruction to EMR G.G.4: Cannot group and instruction that changes bits in SR with a MOVE-like instruction to SR G.G.4: Double pop may only include one register from eeeee or EEEEE G.G.4: Double push may only include one register from eeeee or EEEEE G.G.4: Duplicate destinations within the VLES G.G.4: Only one address register update permitted within a VLES G.G.4: Only one change of flow permitted within group G.G.4: Only one SP update instruction permitted within a VLES G.G.4: Only one T bit update instructions permitted within a VLES G.G.4: Only one VF0-1 or VF2-3 update instruction permitted within a VLES
GG5	G.G.5: DALU register can only be used up to four times per VLES
GP1	G.P.1: No more than two extension words are possible in a VLES
GP3	G.P.3: Only one debug instruction permitted within a VLES G.P.3: Only one di or ei permitted within a VLES G.P.3: Only one doen, doensh, cont, contd, skipls or break permitted within a VLES G.P.3: Only one mark instruction permitted within a VLES G.P.3: Only one stop, wait, or COF permitted within a VLES

Table 2.4 Programming Rule Error Messages (*continued*)

ID	Error Message
GP4	G.P.4: RTE/D or RTERI/D and AGU or IFT/F/A combination not permitted in same group
GP5	G.P.5: N or M used as source reg of MOVE-like instruction more than once within a VLES
GP6	G.P.6: In a set that includes two subsets, each subset can include maximum of 1 AGU and 2 DALU instructions
GP7	G.P.7: The IFA subset, if used, must be the last subset G.P.7: The IFT/IFF/IFFA subgroups cannot all be present in a VLES
GS	G.S: Pairing of non-pairable instructions
GS1	G.S.1: Only one instruction of type 2 can appear in a VLES G.S.1: type 2 instruction can only be grouped with instructions of type 1
GS2	G.S.1: Only one instruction of type 3 can appear in a VLES G.S.1: type 3 instruction can only be grouped with instructions of type 1
GS3	G.S.3: Instruction of type 4 cannot be grouped with any instruction
LC1	L.C.1: Branch to last two VLESs in a loop not allowed
LC2	L.C.2: Change of flow, WAIT, and STOP instructions are not allowed at LA-2 of a long loop
LC3	L.C.3: No conditional branch is allowed in the VLES before the SA or a short loop
LC5	L.C.5: No conditional branches allowed at LA-3 of a long loop
LC7	L.C.7: Destination address of SKIP/BREAK/CONT cannot be in the same loop
LC9	L.C.9: SKIPS/BREAK/CONT target cannot be followed by 2 consecutive VLESs that are both last addresses of loops
LC10	L.C.10: JSR(D)/BSR(D) to LA-2 of long loop or SA of short loop
LC11	L.C.11: For long loops, in is not allowed to have a delayed COF instruction in LA-3
LC12	L.C.12: For short loops, it is not allowed to have a delayed COF instruction at SA-1
LD1	L.D.1: SKIPLS instruction not allowed immediately after DOEN/SH or MOVE-like instruction to any active LCn
LD2	L.D.2: For a long loop, 3 or 4 VLESs are required between DOEN or MOVE-like instructions
LD3	L.D.3: For a short loop, 2 or 1 VLESs are required between DOENSH or MOVE-like instructions

Table 2.4 Programming Rule Error Messages (*continued*)

ID	Error Message
LD5	L.D.5: 1, 2 or more VLESs required between instruction updating any active LCn and CONT/D instruction
LD6	L.D.6: At least 3 VLESs required between instruction updating the active SAn register and LA
LD7	L.D.7: At least 1 VLES required between CONT/D and modification of any active SAn
LD8	L.D.8: At least 3 VLESs required between read of the active LCn and the last VLES of loop
LD9	L.D.9: At least 1 VLES required between read of the active LCn and the first VLES of short loop
LG3	L.G.3: Less than 3 VLESs between MOVE/PUSH SR and end of long loop
LG4	L.G.4: Short loop SA follows MOVE/PUSH SR
LG5	L.G.5: DOEN and loop type mismatch
LL1	L.L.1: Illegal delay slot instruction in last 2 VLESs of a loop
LL2	L.L.2: DOEN/SH or write to any active LCn not allowed in last 3 VLESs of a loop
LL3	L.L.3: Illegal instruction in short loop
LL4	L.L.4: Nested short LOOPEND cannot occur in last 2 VLESs of outer loop
LL5	L.L.5: MOVE-like instruction updating SR is not allowed at LA-4 and LA-3 for long loop
LL6	L.L.6: MOVE-like instruction updating SR is not allowed at SA-2 and SA-1 for short loop
LL7	L.L.7: DOEN/SH and MOVE-like instruction referencing SR/LC register as a whole register is restricted within short loop
LN1	L.N.1: Nested loops can’t end in same address
LN2	L.N.2: A loop may only be nested inside a loop of a smaller DOEN number
LN3	L.N.3: A DOEN/SH instruction or a LOOPENDx pseudo-instruction cannot come between a DOEN/SH instruction and its LOOPSTARTn assembler directive
SR2	SR.2: Instruction may be affected by previous SR change SR.2: Instruction not allowed within two VLESs of a SR change
SR2,3	SR.2,3: violation of SR.2, or 3 restrictions within short/long loop
SR3	SR.3: Instruction not allowed after SR change

Table 2.4 Programming Rule Error Messages (*continued*)

ID	Error Message
SR4	SR.4: There should be two VLESs between a VLES that updates the DOVF in EMR with a MOVE-like instruction to EMR
SR4a	SR.4a: Instructions that affect DOVF bit can't be grouped with a MOVE-like instruction to SR register that clears the OVE bit
SR4,5	SR.4,5: violation of either SR.4 or SR.5 restriction within short/long Loop
SR5	SR.5: Cannot group an instruction that changes the DOVF in EMR with a MOVE-like instruction to EMR
SR7	SR.7: A MOVE-like instruction that reads SR cannot be grouped
T1	T.1: IFc not allowed to follow a group containing a T bit modification T.1: IFc not allowed to follow a group containing a T bit modification within short/long loop
X1	X.1: R0-15, B0-7, N0-3 or M0-3 write in delay slot of COF instruction
X2	X.2: MCTL write in parallel or delay slot of a COF instruction
X3	X.3: T bit modification in parallel to a non-delayed COF or in the delay slot of a COF instruction
X4	X.4: MOVE-like that writes SR just before or in parallel to non-delayed COF or in delay slot or in parallel to delayed COF instruction

NOTE The `-s all` pattern does not include the `Xn` items of Table 2.4, because they are programming guidelines instead of restrictions. To check for violations of these guidelines, you must list them individually, as in the pattern `-s X1,X3,X4`.
If the command line includes the `-arch rainbow` and `-s` all options, the assembler checks for reads and writes to the same address, checks whether accessed-memory bandwidth exceeds 64 bits, and checks for writes on 128 bits.

Data Analysis

The `-k[conv_file]` command-line option activates the 8101 restriction checker, which is appropriate only for the SC140 architecture. All the checked restrictions are part of the 1K42A silicon errata. If the restriction checker finds violations of these restrictions, it issues corresponding error messages. If the -k option does not specify an initialization

Freescale Semiconductor, Inc.
SC_Assem.book Page 29 Thursday, September 23, 2004 10:44 AM

file, the restriction checker cannot check for restriction violations that involve special register addresses.

The directory that contains the assembler also contains a template for the convention file. If the file you are assembling modifies board-dependent registers, you should:

1. De-comment the lines that contain information about these registers.
2. Add the addresses of these registers.

Table 2.5 lists the restrictions that the `-k` option checks.

Table 2.5 SC140 (-k) Restrictions

Identifier	Error Message
CPM6	Spurious overrun indications on the FCC may occur after the stop transmit command is issued.
CPM8	The TODR mechanism may freeze serial channels.
CPM9	Under certain conditions, an ATM receiver may receive cells of PHYs which were not addressed for it.
CPM10	In ATM, a Transmit internal rate underrun error is not reported correctly in the TIRU field of the FCCE register.
CPM14	Corruption of ATM cells may occur when the following combination in used: AAL1 with UDC in which the user-defined header size is 9 to 12 octets and PM is not used.
CPM15	The PDATA, PDATB, PDATC and PDATD registers can only be written with a 32-bit write instruction.
CPM27	The heartbeat checking in FCC transmit ethernet 10 Mbps does not work properly.
CPM28	In the SCC Rx in HDLC mode, RFTHR does not work.
CPM29	In SCC Rx ethernet, the option of transferring only part of a frame into memory does not work.
CPM30	The graceful stop command does not work in SCC Tx in Ethernet, HDLC and Transparent modes.
CPM35	When SCC transparent, envelope mode is used and the received frame size is (4*n)+1, the last byte us corrupted. When GSMR_H(RFW) - rx FIFO width is used, the received data is completely corrupted, not just the last byte.
CPM36	SI's sync signal may not change exactly on clock edge in the following cases: fsd = 00, ce = 0, fe = 0, dsc = 1 fsd = 00, ce = 1, fe = 1, dsc = 1

For More Information: www.freescale.com

Table 2.5 SC140 (-k) Restrictions (*continued*)

Identifier	Error Message
CPM40	There is a rare case when transmitting an ATM idle cell that the idle cell may be corrupted.
CPM41	There are some limitations in the ATM controller. Only the first 8 PM tables can be used and the user must clear the 5 most significant bits of TBD_BASE or RBD_BASE.
CPM42	Data corruption may occur in the receive buffers if MCC channels when more than one TDM slot uses 7 bits of contiguous data.
CPM43	When the FCC transmitter is configured to work in UTOPIA single PHY master mode, it ignores negation of the TxCLAV signal.
CPM44	When the MCC transmitter is used in HDLC super channel mode, a zero insertion at the last bit before the flag fails to occur.
CPM45	In FCC ATM transmit master mode (multiple PHY only), the CLAV signal is sampled 5 clocks before the end of the cell.
CPM46	Each of the communication controllers issues request for service to the CPM with different priorities in order to receive the necessary assistance in time, Because of an internal connection error, the FCC3 request for service is issued with a much lower priority than intended.
CPM48	Disabling TDMx may interfere with the operation of TDMy if TDMy uses the SI-RAM clocks directly above those used by TDMx.
CPM49	External CAM address recognition in Fast Ethernet controller does not function.
CPM50	When the MCC is configured to work in Transparent, super channel first sync slot synchronization, loss of alignment may occur when the first data (idles) on the Tx data line matches the value of the RCVSYNC parameter.
CPM54	Switch may not occur properly when the last entry in either normal or shadow RAM is either one or two bits.
CPM55	The ATM Transmit command does not execute correctly when used on APC priority above 4.
CPM57	The second part of a second cell may overwrite the second part of the first cell in an AAL5 frame.
CPM71	When the MCC performs a DMA read or write of the buffer descriptor GBL is not asserted and TC2 is always driven low.
CPM74	When running an FCC in HDLC nibble mode with the multi-frame per FIFO bit off the CPM may lose synchronization with the FCC HDLC controller.

Table 2.5 SC140 (-k) Restrictions (*continued*)

Identifier	Error Message
CPM76	When using an FCC in transparent mode the first bit of a frame is transmitted as zero every time RTS is asserted before CTS is asserted when CTS is sampled synchronously with data.
CPM82	Dynamic switching in SIRAM may not execute properly.
DMA1	Data transferred by the DMA on either the PPC Bus or Local Bus may be corrupted.
EFC1	When using normal (dual) DMA or fly-by DMA transfers which have maximum transfer size greater than 32-bits with the EFCOP to perform IIR filtering with two or less IIR coefficients, the output of the IIR filter will be lost. The rest of the outputs will be shifted and inaccurate.
LE4	Write to PCTL0 immediately freezes the core for 150-900 cycles. If the system is busy, the core may not exit the freeze state.
LE6	After writing data to the Trace Buffer, the TB is disabled, in order to read from it. Reading the TB by software or from the JTAG may result in incorrect data.
QCPM1	Data may be lost on RETRY when global transactions are performed in 60x compatible mode.
QSIU4	Using Extended mode on the local bus can generate incorrect transactions in certain combinations of consecutive reads and writes.
SIU3	Two consecutive UPM RAM words with REDO != 0 does not execute correctly.
SIU4	MCP due to data errors (parity/ECC) is masked by the SWRI bit in SYPCR.
SIU7	GPCM produces a glitch on the BS lines when seta=1 and scy=0 are used.
SIU8	During a read from a device with port size less than 64 bits, from an address not aligned to 64 bits, the parity bits for parity check are not taken from the correct location.
SIU13	SDAMUX signal is disabled when SDRAM machine handles the memory access and the chip is programmed to single-MPC8260 mode (BCR[EBM]=0).

For maximum effectiveness, the checker must propagate known values of registers. To make this possible:

- For compiled code, you should either:
  - Compile using O0 and defining a volatile extern symbol, that is also provided by the linker command file, or
  - Compile using O3.

- For assembly language code, you should use EQUs (visible in the file submitted for checks) for initializing special registers involved in accessing peripherals.
- For variables whose values cannot be known until the linking stage, you should use the `.value` directive.
- If you use the `-c` or `-k` options when you assemble, do not begin start or basic-block markers with the letters DW. (The assembler would treat such markers as debug labels that the compiler introduces.)

Code Examples

Listing 2.1 shows C code appropriate for -O0 compilation. Listing 2.2 shows C code appropriate for -O3 compilation. Listing 2.3 shows assembly code.

Listing 2.1 C Example for -O0 Compilation

```
volatile extern int bcr;
volatile extern int psdmr;

void initialize()
{
      bcr = 0x10000000; /*EBM = 0*/
    psdmr = 0x90000000; /*EAMUX = 1*/
}

void main()
{
      initialize(); /*SIU13 violation*/
}
```

Listing 2.2 C Example for -O3 Compilation

.h file:

```
#define BR0 0x100
#define MAR 0x20
```

.c file:

```
void initialize
{
      int *br;

      br = (int *)BR0;
```

```
      *br = 0x280000; /*PS,DECC != 0*/
}

void initialize_read_from_UPM(int addr)
{
      int *mar;

      mar = (int *)MAR;
      *mar = addr;
}

void main()
{
      initialize_UPM();
      initialize_read_from_UPM(4); /*ADDRESS = 4*/ /*SIU 8 violation*/
}
```

Listing 2.3 Assembly Language Code

```
      move.l #$6260,r0  ;si2cmr
      move.l #$1900,d0  ;rfsd = tfsd = 0, ce = fe = 1, dsc = 1
      move d0,(r0)      ;CPM36 violation
```

Data Analysis Terms

These terms apply to the assembler's data analyzer:

- **Calling convention** — convention the restriction checker uses for external calls. This convention specifies the registers and memory addresses that the called routine affects.
- **Call tree** — list of calling routines and called routines, from a source file, depicted in tree form.
- **Constant propagation** — algorithm that propagates constant resources over a control-flow graph.
- **Control-flow graph (CFG)** — rooted, directed graph that provides information about the flow of a routine.

Data Analysis Limitations

Keep in mind these limitations for -k restriction checking:

- The stack content is not available to the checker, nor can the checker know the initial values of the (O)SP registers.

- The checker treats PUSHN and POPN instructions the same way it treats regular PUSH/POP instructions.
- The checker always operates as if the processor is in the normal processing state: SP is NSP and OSP is ESP.
- Two code sections in the same asm file prevents restriction checking. (The assembler starts both such sections at p:$0, letting the linker choose the real addresses. This means that at least two instructions start at p:$0.)
- Peripherals must have the same endianness as the assembled file.
- The checker cannot distinguish a call to address p:$0 from a call to an external routine.

Initialization File

The restriction checker relies on the initialization file for address definitions, label values, and calling conventions. You also can use this file to specify the addresses of such special registers as Brx and ORx (for memory controllers).

To include comments in the initialization file, write # as the first character of each comment line.

These directives may appear in the initialization file:

.address `register_name mem_address`

Assigns the specified hexadecimal address to the specified register; the register *must* be a memory mapped register. Neither argument is case sensitive. The `mem_address` value has the format `p:xxxxxxxx`.

Example: `.address Br0 p:ff801801`

.call-conv `call_conv_id`
[deleted]=`[list_of_deleted_regs_and_mem_addresses]`

Defines a calling convention. The `call_conv_id` name may be any ASCII string, except that the restriction checker ignores duplicate names.

The list in brackets consists of registers and hexadecimal memory intervals, separated by commas. Register names must have `r.reg_` prefixes, such as `r:reg_r0` or `r:reg_n3`. Memory intervals must follow the format `p:100-300`. To specify a single address, follow the format `p:200` or `p:200-200`.

Example:

```
.call_conv 1
      deleted=[r:reg_d0,r:reg_sp,p:ffff]
      deleted=[r:reg_r1,r:reg_r2]
```

.funcName=call_conv_id

Specifies the calling convention for a function.

Example: `.func_fibonacci=1`

If the restriction checker finds a call to an external routine, the checker searches the initialization file's list of function calling conventions:

- If the list includes the function name, the checker applies the specified calling convention.
- If the list does not include the function name, the checker uses the default convention: the called function changes all the core registers without affecting memory addresses.

As the constant propagation algorithm needs an existing CFG, indirect change-of-flow instructions should have extra target specifications. The compiler may provide this information. But manually assembled files should pass this information to the checker in the form of a comment. This comment should be either on the line of the COF instruction or the last line of the packet that contains the COF instruction.

In this sample code, such a comment includes the symbols `l1` and `l2`, the possible targets of the jump instruction:

```
[
    move.w #$1234,d0
    jsrd (r0)
]                           ;Lint_info: targets: l1 l2
```

`.never_return_symbols` `list_of_symbols`

Tells assembler to *not* return to the current function, if a conditional or unconditional jump instruction hits any external symbol of the input list. Pertains to such instructions as `bra`, `break`, `cont`, `contd`, and `jmp`. (You can use this directive to make sure that the current basic block will not have any successors.)

Example: `.never_return_symbols _abort,___QCtxtRestore`

`.value` `label value`

Assigns the specified hexadecimal value to the label; the label may be a symbol known during the linking stage. The label is case sensitive.

Example: `.value StArT ffff`

Assembler Processing

The SC100 assembler passes through code three times, performing these operations:

1. First Pass:
   - Gathers instruction sequence and ordering information.
   - As appropriate, rearranges instructions, generates error messages and warnings.

2. Second Pass:
   - Reads source program.
   - Builds symbol and macro tables.
3. Third Pass:
   - Referring to the Pass 2 tables, generates the object file.
   - Produces the source listing.

The assembler processes each source statement completely before reading the next statement. As it reads each line, the assembler applies all translations that DEFINE directives specify. Then the assembler examines the label, operation code, and operand fields. The assembler scans the macro definition table for matches with the operation code. If there is no match, the assembler scans the operation code and directive tables for matches with known opcode.

In case of an error, the assembler displays the appropriate error message, then the line that contains the error. The assembler displays all error messages, even if it does not generate a source listing. At the end of the source listing, the assembler prints error, warning, and remark totals. When the assembler returns control to the host operating system, it returns the number of errors as an exit status.

Source Statements

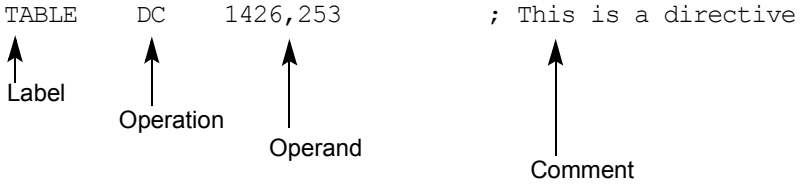
Assembly language programs consist of two types of source statements:

- Assembly language instructions and a comment field
- An assembler directive and a comment field.

The SC140 and SC100 assembly language supports conditional assembly. It also supports macros that replace a single program statement with the statements of the macro definition.

Figure 2.1 shows the four fields of the simplest source statement: label, operation, operand, and comment. Later sections of this chapter explain each field.

Figure 2.1 Basic Source Statement



Spaces or tabs must separate fields. The label, operation, and operand fields must not include spaces, except for spaces in quoted strings.

Only the first three fields are significant for the assembler; it ignores the comment field. The assembler treats anything beginning in column 1 as a label.

To extend a source statement to multiple lines, end all but the last line with the continuation character (\). Exception: An instruction group can span multiple lines without continuation characters, provided that brackets ([]) enclose the group.

The maximum length for a source statement, including any continuation lines is 4000 characters. Assembler mnemonics and directives are not case sensitive. But case does matter for labels, symbols, directive arguments, and literal strings.

If the source file contains horizontal tab characters (ASCII $09), the assembler moves them to the next fixed tab stop. The default stops are at eight-character intervals: columns 1, 9, 17, and so forth, but you can use the TAB directive to change the stops.

Label Field

Labels begin in column 1 of a source statement. If a line's first character is a space or tab, it probably means that the label field is empty. Label rules are:

- Label names must follow the same conventions as symbol names.
- A label whose first character is an underscore (_) is a *global label.*
- To indent a label, end it with a colon (:). Only space or tab characters may precede such an indented label.
- A label may occur only once in the label field of an individual source file, unless it is a local label or is used with the SET directive. If any non-local label occurs more than once in a label field, the assembler flags all references but the first as errors.
- A line may consist of only a label. Such a line assigns the value of the location counter to the label. Except for some directives, the assembler assigns a label the location-counter value for the first word of the instruction or data being assembled.

Operation Field

The operation field follows the label field; at least one space or tab must precede the operation field. Operation-field entries may be:

- Opcodes — Mnemonics that correspond directly to DSP machine instructions.
- Directives — Special assembler operation codes that control the assembly process.
- Macro calls — Invocations or macros, already defined.

The assembler first searches for operation codes in an internal macro definition table. If it does not find a match, it searches the table of machine operation codes and assembler

directives. If neither of the tables holds the specified operation code, the assembler generates an error message.

To change this sequence, you can use the MACLIB directive. This means that macro names can replace standard machine operation codes and assembler directives, although the assembler issues warnings about such replacements.

Operand Field

The effect of the operand field depends on the contents of the operation field. Any operand-field value must follow the operation field; at least one space or tab must precede the operand value. Operand values may include symbols, expressions, or a combination of both; commas without spaces must separate multiple symbols or expressions.

As well as an operand value, the operand field includes the addressing mode for the instruction. For addressing mode definitions, see the core reference manual for your processor.

Comment Field

The assembler ignores comments, but you should include them in your source files for internal documentation. A comment field consists of a semicolon (;), followed by any characters that are not part of a literal string.

If a comment starts in the first column of the source file, the assembler aligns it with the label field. Otherwise, the assembler aligns comments the comment field. To prevent comments' reproduction in the source listing (or to prevent them being saved in macro definitions), start the comments with two successive semicolons (;;).

Variable Length Execution Sets

The SC100 architecture supports variable length execution sets (VLESes): grouping multiple instructions for parallel execution. For VLES grouping and sequencing rules, see the core reference manual for your processor.

The assembler interprets each line containing instructions as a VLES. Tabs or spaces must separate instructions, as this example shows:

```
move.f  r2)+,d0  move.f  r2)+,d8  clr d5  ;VLES, 3
instructions
```

To have a VLES span several lines, use bracket delimiters ([ ]), as this example shows:

```
[
  mac d0,d1,d2        mac d3,d4,d5        ; multiply
operands
  add d0,d1,d3        add d3,d4,d6        ; add operands
```

```
  move.f (r0)+,d0     move.w (r1)+,d1     ; load new
operands
]
```

NOTE Lines of this example include only two instructions and one comment. This practice improves readability; but it is not required.
You should separate DALU and AGU instructions in a VLES: start with DALU instructions and end with AGU instructions.

Symbol Names

Follow these conventions for symbol names:

- Names can be from one to 4000 characters long.
- Names cannot begin with number characters 0—9. Otherwise, names can be any combination of alphanumeric characters (A—Z, a—z, 0—9) and the underscore character (_).
- Names and other identifiers containing a period (.) are reserved for the system.
- Names are case sensitive, but you can use the -oIC option to override the distinction between upper-case and lower-case letters.
- Names, regardless of case, cannot duplicate the names of SC100 core registers. The assembler reserves these register names.
- Names must not duplicate the names of registers, instructions, or pseudo-instructions. The assembler reserves these names.

This table shows examples of symbol names:

Type	Example
Valid names	loop_1; ENTRY; _alpha_BRAVO_charlie
Invalid names	1st_loop; loop&go; $value
Reserved names	loop.elf; .loop; Loop.Go; move; dcl; r0

Symbol Labels

You may use symbols as labels. To make a label local, start it with the percent character (%). This limits the label's scope to the area between any two non-local labels. The only source statements that can refer to or define such a local label are the statements between the source lines that contain the non-local labels. A local label is useful as the terminating

address of a DO loop, or any such location that must have a unique label, but is not significant for documenting the source file.

In a macro, however, the scope of local labels is the entire macro expansion, without regard to non-local labels. Accordingly, all local labels within a macro must be unique. You can use such local labels freely within a macro definition, without regard to the number of macro expansions.

The assembler treats non-local labels within a macro expansion as normal labels. This means that such labels cannot occur more than once, unless you use them with the SET directive.

Strings

Literal ASCII strings can be operands for some assembler directives; they also have limited use in expressions. Such a string is one or more ASCII characters enclosed by single quotes ('). To specify an apostrophe within a literal string, use two consecutive apostrophe characters.

The alternate string delimiter is the double quote (") character. If you use double quotes to enclose a string, the assembler expands any DEFINE directive symbols contained in the string.

NOTE Be careful about using the double-quote character inside macros, where this character is a dummy argument string operator. You can use the macro concatenation operator to escape a double-quoted string.

The concatenation operator (++) tells the assembler to consider two strings to be one. For example, the strings `'ABC'++'DEF'` and `'ABCDEF'` are identical to the assembler.

Use brackets ([ ]) to have the assembler extract a substring. For example, if the assembler encounters the expression `['abcdefg',1,3]`, it uses the string value `'bcd'`. Substrings are valid wherever strings are, and you can nest substrings.

The assembler includes functions for determining the length of a string, and the position of one string within another.

Source Listing

The source listing consists of the original source statements, formatted for easier reading, as well as other information the assembler generates. Most listing lines correspond directly to a source statement. Listing lines that do not correspond directly to source statements are page headings, error messages, expansions of macro calls, or expansions of directives such as DC.

According to its default setting, the assembler sends the source listing to the standard output. Options are:

- Sending the source listing to a printer, file, null device, or other such arbitrary destination. For this option, use the I/O redirection facilities of the host operating system.
- Sending the source listing to an argument file of the `-l` command-line option. If the `-l` option lacks an argument file, the assembler creates a source listing. To name this listing, the assembler adds the `.lst` extension to the name of the first source file in the command line.
- Inhibiting the source listing, by using the `IL` (inhibit listing) option.

Note that the `-b` and `-l` command-line options allow a hyphen as an argument: this directs the corresponding output to the standard output stream. But unpredictable results may occur if your settings send both the object file and the source listing to the same output stream.

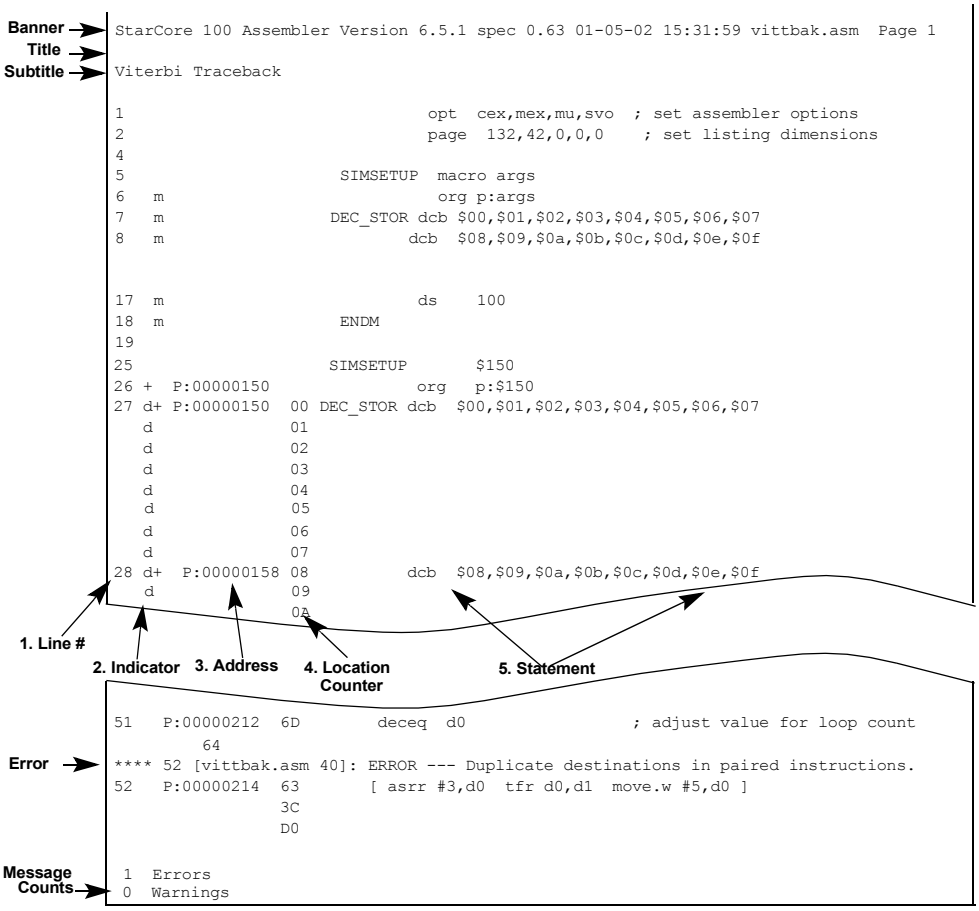
The assembler always sends error messages to the standard output, regardless of option settings.

Source Listing Example

Figure 2.2 shows an example source listing. Text immediately after the figure explains areas of interest:

SC100 Assembler
Source Listing

Figure 2.2 Assembler Source Listing



Areas of interest are:

- Banner — The first line of each page. The banner consists of the assembler and version number, the date and time of assembly, the source file name, and the listing page number.
- Titles — Line 2 displays the title and line 3 displays the subtitle, provided that you have defined these titles. (Use the TITLE and STITLE directives.) If you have not defined titles, these lines are blank.
- Line number — The first field of source listing lines shows the line number.

For More Information: www.freescale.com

- Indicator — The second field of source listing lines is the macro definition/expansion column. Possible values are:

Indicator	Meaning
`m`	Macro definition in progress. (The assembler does not assemble these lines, but retains them for macro expansion.)
`+`	Macro expansion in progress.
`d`	Data expansion occurring. (The `-oCEX` option requested this expansion.)
`i`	Line skipped due to an IF-THEN-ELSE directive sequence.
`p`	Line automatically generated by the assembler, for a pipeline delay.

- Address — The third field of source listing lines contains the memory space value.
- Location Counter — The fourth field of source listing lines contains the location counter value.
- Statement — Fields 5 and beyond of source listing lines contain the source statement. This statement contains one or more instructions, depending on usage of instruction groups.
- Error Message — The listing shows an error message above the line that contains the error. The message consists of the source file name, the source line number, the severity level (remark, warning, error, or fatal), and the message text. The message may also include information about incorrect symbols or fields.
- Message Counts — The listing ends with counts of the assembler errors and warnings.

Final Information

The end of a source listing includes additional information that the Assembler records, such as:

- A list of macros, including the definition line numbers. This list includes any sections declared in the program, as well as defined symbols and their substitution strings.
- A symbol table, including symbol types (integer or floating point), values, definition sections, and attributes (local, global, or set).
- An optional table of non-local symbols, with their defined values. (Use the `S` option to have the assembler create this table.) This table appears after the end of source input or after the END directive, whichever is first.

- An optional cross-reference table of non-local symbols, with line numbers of source lines that refer to these symbols. Asterisk (*) characters indicate lines that define symbols. (Use the `CRE` option to have the assembler create this table.) This table appears after the end of source input or after the END directive, whichever is first.
- An optional memory utilization report: a memory map showing data allocation, code generation, and unused memory areas. If the information is available, this report also shows associated labels and section information. (Use the `MU` option to have the assembler create this table.) This table appears at the end of the source listing. (The Type column indicates the kind of generated code or data, or indicates that a block is unused. The type DATA denotes uninitialized data areas; other types denote initialized data, code, and reverse carry buffers.)

3

Expressions

An expression represents a value that can be an operand of an assembler instruction or directive. Expressions consist of symbols, constants, operators, and parentheses. Expressions may contain:

- User-defined labels, with their integer or floating-point values
- Integers
- Floating-point numbers
- ASCII literal strings

In general, you may not use space or tab characters between the terms and operators of assembler expressions. Otherwise, expressions follow the rules of algebra and boolean arithmetic.

Absolute and Relative Expressions

If the assembler operates in relative mode, all address expressions must follow these definitions:

- **Absolute expression** — An expression that consists only of absolute terms, or is the result of two relative terms with opposing signs.
- **Relative expression** — An expression that consists of a relative term by itself or a relative term in combination with absolute terms.

Only these types of expressions retain meaningful values after program relocation. For example, if your program pairs relative terms with opposing signs, the result is the difference between the two relative terms — an absolute value. But if code adds together two positive relative terms, the result is unpredictable — it depends on the terms' computed values at relocation time.

Expression Memory Space Attributes

As the assembler evaluates an expression, it uses the associated integer or floating-point value in place of each expression symbol. Each symbol also includes a memory space attribute: P (program) or N (none).

The result of an expression always has an associated memory space attribute:

For More Information: www.freescale.com

- Label, constant, and floating-point expressions associated with the SET directive always have the memory space attribute N.
- The unary logical negate operator, relational operators, logical operators, and some functions return values that have the memory space attribute N.
- The result of an expression that has only one operand (and possibly the unary negate or unary minus operator) always has the memory attribute of that operand.
- The results of expressions involving operands with different memory space attributes have the memory space attribute P.

The assembler treats the memory space attribute as a type, as high-level languages use type for variables. Symbols that have the memory space attribute P should be addresses, so their maximum values should not exceed the maximum address value of the DSP inclusive. Only symbols that have the memory space attribute N can have values greater than the target processor's maximum address.

The memory space is implicitly P if you use an address as the operand of a LOOP, branch, or jump-type instruction.

Immediate addressing expressions can have any memory space attribute.

Internal Expression Representation

The assembler's internal representation of expression values depends on the target-processor word size. The assembler supports word and double-word integer formats. Although the actual storage size of an expression value depends on the result's magnitude, the assembler can represent signed integers as long as 64 bits.

Internal floating-point representation depends almost entirely on the host environment, but the assembler's usual storage format for floating-point values is double precision. This format consists of 64 bits: 53 bits for the mantissa, 11 bits for the exponent, and an implied binary point.

Constants

Constants represent data values that do not vary during program execution.

Numeric Constants

Table 3.1 explains the possible numeric constants.

Table 3.1 Numeric Constants

Type	Description	Examples
Binary	Percent sign (%) followed by string of binary digits (0,1)	%11010
Hexadecimal	Dollar sign ($) followed by string of hexadecimal digits (0-9, A-F, a-f)	$12FF $12ff
Decimal integer	String of decimal digits (0-9). Optional grave accent (`) can start the string.	12345
Decimal floating point	String of decimal digits that includes a decimal point or the letter E. The digits after the letter E are the exponent.	6E10 .6 2.7e2

The default radix is 10, so the assembler treats constants without leading radixes as decimal values. If you wish, you can use the RADIX directive to change the default input radix: setting the input radix to 16 would let you omit the dollar sign ($) indicator from hexadecimal constants.

String Constants

The assembler converts expression string constants to right-aligned, concatenated sequences of ASCII bytes. Null strings have the value 0.

Examples are:

```
'ABCD'        ($41424344)
'''79'        ($00273739)
'A'           ($00000041)
''            ($00000000)  ← null string
'abcdef'      ($61626364)
'abc'++'de'   ($61626364)
```

The size limit for string expressions is the long-word size of the target processor. If a string exceeds this number of characters, the assembler truncates the value and prints a warning. This restriction also applies to string constants that involve the string concatenation operator, except for the DC and DCB directives.

Operators

Most assembler operators pertain to both floating-point and integer values. The assembler follows these rules:

- If both operands are integers, the result is an integer value.
- If both operands are floating point values, the result is a floating-point value.
- If one operand is a floating-point value and the other operand is an integer value, the assembler converts the integer to a floating-point value, then applies the operator. The result is a floating-point value.

Table 3.2 explains assembler operators, noting those restricted to integer operands. The main use of the relational and logical operands is with the IF conditional-assembly directive, although you can use these operands in any expression.

Table 3.2 Assembler Operators

Type	Operator	Description
Unary	+	Plus — Returns the positive value of its operand.
	-	Minus — Returns the negative value of its operand.
	~	One's Complement — Return's the one's complement of its integer operand; cannot be used with floating-point operands.
	!	Logical Negate — Returns an integer 1 if the value of its operand is 0; otherwise returns a 0. The result's memory space attribute is N Example: If symbol `BUF` has the value 0, `!BUF` has the value 1. If `BUF` has the value 1000, `!BUF` has the value 0.
Arithmetic	+	Addition — Yields the sum of its operands.
	-	Subtraction — Yields the difference between its operands.
	*	Multiplication — Yields the product of its operands.
	/	Division — Yields the quotient: the first operand divided by the second. For integer operands, the result is a truncated integer.
	%	Mod — Yields the remainder of the first operand divided by the second. (If both operands are floating-point values and the divisor is 0.0, the result is the dividend.)

Table 3.2 Assembler Operators (*continued*)

Type	Operator	Description
Shift	<<	Shift Left — For integer operands only. Shifts and zero fills the left operand to the left; the right operand specifies the number of bits to shift.
	>>	Shift Right — For integer operands only. Shifts the left operand to the right; the right operand specifies the number of bits to shift. Extends the sign bit.
Relational	<	Less Than — Returns integer 1 if the expression is true, integer 0 if the expression is false. The result's memory space attribute is N. Example: If D = 3 and E = 5, D < E = 1.
	>	Greater Than — Returns integer 1 if the expression is true, integer 0 if the expression is false. The result's memory space attribute is N. Example: If D = 3 and E = 5, D > E = 0.
	<=	Less Than or Equal — Returns integer 1 if the expression is true, integer 0 if the expression is false. The result's memory space attribute is N.
	>=	Greater Than or Equal — Returns integer 1 if the expression is true, integer 0 if the expression is false. The result's memory space attribute is N.
	==	Equal — Returns integer 1 if the expression is true, integer 0 if the expression is false. The result's memory space attribute is N.
	!=	Not Equal — Returns integer 1 if the expression is false, integer 0 if the expression is true. The result's memory space attribute is N.
Bitwise	&	AND — For integers only. Yields the bitwise AND function of its operands.
	|	OR — For integers only. Yields the bitwise OR function of its operands.
	^	Exclusive OR — For integers only. Yields the bitwise exclusive OR function of its operands.

Table 3.2 Assembler Operators (*continued*)

Type	Operator	Description
Logical	&&	Logical AND — Returns integer 1 if both operands are nonzero; otherwise returns integer 0.
	||	Logical OR — Returns integer 1 if either operand is nonzero; otherwise returns integer 0.

Operator Precedence

The assembler evaluates expressions from left to right, following these rules of operator precedence:

1. Parenthetical expression (innermost first)
2. Unary plus, unary minus, one's complement, logical negation
3. Multiplication, division, mod
4. Addition, subtraction
5. Shift
6. Relational operators: less, less or equal, greater, greater or equal
7. Relational operators: equal, not equal
8. Bitwise AND, OR, exclusive OR
9. Logical AND, OR

Valid operands include numeric constants, literal ASCII strings, and symbols.

You cannot apply the one's complement, shift, or bitwise operators to floating-point operands. That is, if an expression evaluation results in a floating-point value on either side of any such operator, the assembler generates an error message.

Functions

The assembler's built-in functions support data conversion, string comparison, and transcendental math computations. You may use functions as terms in any arbitrary expression; functions may have no arguments, one argument, or multiple arguments. These rules apply:

- Open and close parentheses must always follow functions.
- Arguments that are expressions must be absolute expressions, except where noted.
- Arguments must not contain external references.

- There must not be intervening spaces between the function name and the open parenthesis, or between comma-separated arguments.

Table 3.3 lists the assembler functions of each type:

Table 3.3 Assembler Function List

Type	Function
Mathematical	ABS — Absolute value
	ACS — Arc cosine
	ASN — Arc sine
	AT2 — Arc tangent
	ATN — Arc tangent
	CEL — Ceiling
	COH — Hyperbolic cosine
	COS — Cosine
	FLR — Floor
	L10 — Log base 10
	LOG — Natural logarithm
	MAX — Maximum value
	MIN — Minimum value
	POW — Raise to power
	RND — Random value
	SGN — Return sign
	SIN — Sine
	SNH — Hyperbolic sine
	SQT — Square root
	TAN — Tangent
	TNH — Hyperbolic tangent
	XPN — Exponential function

Table 3.3 Assembler Function List (*continued*)

Type	Function
Conversion	CVF — Convert Integer to floating point
	CVI — Convert floating point to integer
	CVS — Convert Memory space
	FLD — Shift and mask
	FRC — Convert floating point to fractional
	LFR — Convert floating point to long fractional
	LNG — Concatenate to double word
	LUN — Convert long fractional to floating point
	RVB — Reverse bits in field
	UNF — Convert fractional to floating point
String	LEN — String length
	POS — Position of substring
	SCP — Compare strings
Macro	ARG — Macro argument
	CNT — Macro argument count
	MAC — Macro definition
	MXP — Macro expansion
Assembler Mode	BIGENDIAN — Endian mode check
	CCC — Cumulative cycle count
	CHK — Instruction/data checksum
	CTR — Location counter number
	DEF — Defined symbol
	EXP — Expression check
	INT — Integer check
	LCV — Location counter value

Table 3.3 Assembler Function List (*continued*)

Type	Function
	LST — LIST directive flag value
	MSP — Memory space
	REL — Relative mode

Descriptions of the assembler functions complete this chapter. These descriptions are in alphabetic order, without regard to function types. Although these descriptions show functions in upper case, the functions are not case sensitive.

ABS Absolute Value

Returns the absolute value of the specified expression, as a floating-point value. The result's memory space attribute is N.

```
@ABS(expr)
```

Parameter

`expr`

Any valid expression.

Example:

```
MOVE #@ABS(VAL),D4.S      ; Load absolute value
```

ACS Arc Cosine

Returns the arc cosine of the specified expression, as a floating-point value, in the range zero to pi. The result's memory space attribute is N.

```
@ACS(expr)
```

Parameter

`expr`

Any valid expression that evaluates to a value between -1 and 1.

For More Information: www.freescale.com

Example:

```
ACOS = @ACS(-1.0)       ; ACOS = 3.141593
```

ARG Macro Argument

Returns integer 1 if the specified macro argument is present; otherwise returns 0. The result's memory space attribute is N.

```
@ARG(symbol | expr)
```

Parameters

`symbol`

Any valid symbol that refers to a dummy argument name; must be in quotes.

`expr`

Any valid expression that refers to the argument's ordinal position in the macro dummy argument list.

Remarks

If you use this function when no macro expansion is active, the assembler issues a warning.

Example:

```
IF   @ARG(TWIDDLE)      ; Is twiddle factor provided?
```

ASN Arc Sine

Returns the arc sine of the specified expression, as a floating-point value, in the range -pi/2 to pi/2. The result's memory space attribute is N.

```
@ASN(expr)
```

Parameter

`expr`

Any valid expression that evaluates to a value between -1 and 1.

Example:

```
ARCSINE SET @ASN(-1.0)      ; ARCSINE = -1.570796
```

AT2 Arc Tangent

Returns the arc tangent of the quotient of two expressions (`expr1/expr2`), as a floating-point value, in the range -pi to pi. A comma must separate the `expr1` and `expr2` expressions. The result's memory space attribute is N.

```
@AT2(expr1,expr2)
```

Parameters

`expr1, expr2`

Any valid expressions.

Example:

```
ATAN EQU @AT2(-1.0,1.0      ; ATAN = -0.7853982
```

ATN Arc Tangent

Returns the arc tangent of the specified expression, as a floating-point value, in the range -pi/2 to pi/2. The result's memory space attribute is N.

```
@ATN(expr)
```

Parameter

`expr`

Any valid expression.

Example:

```
MOVE  #@ATN(1.0),D0.S      ; Load arc tangent
```

BIGENDIAN Endian Mode Check

Returns an integer 1 if big-endian mode is enabled; returns a 0 if the default little-endian mode is enabled. The result's memory space attribute is N.

```
@BIGENDIAN()
```

Example:

```
IF @BIGENDIAN()
   DCB "BIG-ENDIAN"
ELSE
   DCB "LITTLE-ENDIAN"
ENDIF
```

NOTE Another way to check for big-endian compiling of an asm file is to use @DEF:

```
IF @DEF('__BIG_ENDIAN__')
  DCB 1
ELSE
  DCB 0
ENDIF
```

CCC Cumulative Cycle Count

Returns the cumulative cycle count as an integer; useful with the CC, NOCC, and CONTC assembler options. The result's memory space attribute is N.

```
@CCC()
```

Example:

```
IF  @CCC() > 200      ; Check if cycle count > 200
```

CEL Ceiling

Returns the ceiling of the specified expression: a floating-point value that represents the smallest integer greater than or equal to the expression. The result's memory space attribute is N.

```
@CEL(expr)
```

Parameter

`expr`

Any valid expression.

Example:

```
CEIL SET @CEL(-1.05)      ; CEIL = -1.0
```

CHK Instruction/Data Checksum

Returns the current instruction/data checksum value as an integer. The result's memory space attribute is N.

```
@CHK()
```

Remarks

Useful in conjunction with the CK, NOCK, and CONTCK assembler options . Note that using directives other than SET to assign the checksum value could lead to phasing errors, due to different generated instruction values between passes.

Example:

```
CHKSUM SET @CHK()       ; Reserve checksum value
```

CNT Macro Argument Count

Returns the count of the current macro expansion arguments as an integer. If you use this function when no macro expansion is active, the assembler issues a warning. The result's memory space attribute is N.

```
@CNT()
```

Example:

```
ARGCNT SET @CNT()       ; Reserve arg count
```

COH Hyperbolic Cosine

Returns the hyperbolic cosine of the specified expression, as a floating-point value. The result's memory space attribute is N.

```
@COH(expr)
```

Parameter

`expr`

Any valid expression.

For More Information: www.freescale.com

Example:

```
HYCOS EQU @COH(VAL)      ; Compute hyperbolic cosine
```

COS Cosine

Returns the cosine of the specified expression, as a floating-point value. The result's memory space attribute is N.

`@COS(expr)`

Parameter

`expr`

Any valid expression.

Example:

```
DC -@COS(@CVF(COUNT)*FREQ)      ; Compute cosine value
```

CTR Location Counter Number

Returns the counter number of the specified location counter. The returned counter number is an integer value with memory space attribute N.

`@CTR(L|R)`

Parameters

`L`

Specifier for the load location counter.

`R`

Specifier for the runtime location counter.

Example:

```
CNUM = @CTR(R)      ; Runtime counter number
```

CVF Convert Integer to Floating Point

Converts the value of the specified expression to a floating-point value. The result's memory space attribute is N.

```
@CVF(expr)
```

Parameter

`expr`

Any valid integer expression.

Example:

```
FLOAT SET @CVF(5)       ; FLOAT = 5.0
```

CVI Convert Floating Point to Integer

Converts the value of the specified expression to an integer value. The result's memory space attribute is N. (Such conversions can be inexact, possibly truncating floating-point values.)

```
@CVI(expr)
```

Parameter

`expr`

Any valid floating-point expression.

Example:

```
INT SET @CVI(-1.05)       ; INT = -1
```

CVS Convert Memory Space

Assigns the specified memory space attribute to the specified expression, returning the same expression.

```
@CVS(P | N,expr)
```


For More Information: www.freescale.com

Parameters

`P`

Specifier for memory space attribute P.

`N`

Specifier for memory space attribute N.

`expr`

Any valid relative or absolute expression.

Example:

```
LOADDR EQU @CVS(P,TARGET)      ; Set LOADDR to P:TARGET
```

DEF Defined Symbol

Returns an integer 1 if the specified symbol is defined; otherwise returns a 0. The result's memory space attribute is N.

`@DEF(symbol)`

Parameter

`symbol`

Any label not associated with a MACRO or SECTION directive. Quotes tell the assembler to look for a DEFINE symbol; if `symbol` is not in quotes, the assembler looks for an ordinary label.

Example:

```
IF  @DEF(ANGLE)      ; Assemble if ANGLE is defined
```

EXP Expression Check

Returns an integer 1 if evaluating the specified expression would result in errors; otherwise returns 0. The result's memory space attribute is N. The assembler does not issue an error message if the expression contains an error; the assembler does not test for warnings.

`@EXP(expr)`

Parameter

`expr`

Any valid relative or absolute expression.

Example:

```
IF  !@EXP(@FRC(VAL))      ; Skip on error
```

FLD Shift and Mask

Shifts and masks the value expression into the base expression for width bits, beginning at the start bit. If you omit the start-bit value, the assembler uses zero (the least significant bit). Returns the shifted and masked value, with memory space attribute N.

`@FLD(base,value,width[,start])`

Parameters

`base`

Original positive-integer expression; may not exceed the target word size.

`value`

Positive-integer expression shifted and masked into `base`; may not exceed the target word size.

`width`

Number of bits to shift; a positive-integer expression that may not exceed the target word size.

`start`

Optional: Starting bit for the operation; a positive-integer expression that may not exceed the target word size. .

Example:

```
SWITCH EQU @FLD(TOG,1,1,7)      ; Turn eighth bit on
```

FLR Floor

Returns the floor of the specified expression: a floating-point value that represents the largest integer less than or equal to the expression. The result's memory space attribute is N.

```
@FLR(expr)
```

Parameter

`expr`

Any valid expression.

Example:

```
FLOOR SET @FLR(2.5)       ; FLOOR  =  2.0
```

FRC Convert Floating Point to Fractional

Scales and convergent rounds a floating-point expression, returning its fractional representation as an integer. The result's memory space attribute is N.

```
@FRC(expr)
```

Parameter

`expr`

Any valid floating-point expression.

Example:

```
FRAC EQU @FRC(FLT)+1      ; Compute saturation
```

INT Integer Check

Returns an integer 1 if the specified expression evaluates to an integer; otherwise returns a 0. The result's memory space attribute is N.

```
@INT(expr)
```

Parameter

`expr`

Any valid relative or absolute expression.

Example:

```
IF  @INT(TERM)      ; Insure integer value
```

L10 Log Base 10

Returns the base 10 logarithm of the specified expression, as a floating-point value. The result's memory space attribute is N.

`@L10(expr)`

Parameter

`expr`

A numerical expression greater than zero.

Example:

```
LOG EQU @L10(100.0)       ; LOG = 2
```

LCV Location Counter Value

Returns the memory space attribute and value of the specified location counter. The optional second argument indicates the Low, High, or numbered counter; a comma must separate the two arguments. If you omit the second argument, the assembler uses the default counter (counter 0).

`@LCV({L | R}[,{L | H | expr}])`

Parameters

`L`

If the first argument, specifier for the load location counter.

If the optional second argument, specifier for the low counter.

`R`

Specifier for the runtime location counter.

`H`

Specifier for the high counter.

`expr`

Specifier for a numbered counter; must evaluate to an integer value.

Remarks

This function does not work correctly if you use it to specify the runtime counter value of a relocatable overlay. This is because the resulting value is an overlay expression, and you may not use overlay expressions to set the runtime counter for a subsequent overlay.

Example:

```
ADDR = @LCV(R)       ; Save runtime address
```

LEN String Length

Returns the length of the specified string, as an integer. The result's memory space attribute is N.

`@LEN(string)`

Parameter

`string`

Any valid string.

Example:

```
SLEN SET @LEN('string')       ; SLEN = 6
```

LFR Convert Floating Point to Long Fractional

Scales and convergent rounds a floating-point expression, returning its fractional representation as a long integer. The result's memory space attribute is N.

`@LFR(expr)`

Parameter

`expr`

Any valid, floating-point expression.

Example:

```
LFRAC EQU @LFR(LFLT)      ; Store binary form
```

LNG Concatenate to Double Word

Concatenates single words into a double word: `expr1` becomes the high word, `expr2` becomes the low word. The result's memory space attribute is N.

`@LNG(expr1,expr2)`

Parameters

`expr1, expr2`

Any valid, single-word expressions.

Example:

```
LWORD DC @LNG(HI,LO)      ; Build long word
```

LOG Natural Logarithm

Returns the natural logarithm of the specified expression, as a floating-point value. The result's memory space attribute is N.

`@LOG(expr)`

Parameter

`expr`

Any valid expression greater than zero.

Example:

```
LOG EQU @LOG(100.0)       ; LOG = 4.605170
```

LST LIST Directive Flag Value

Returns the value of the LIST directive flag as an integer, with memory space attribute N. (Each time the assembler encounters the LIST directive in source code, it increments the flag; each time it encounters the NOLIST directive, it decrements the flag.)

```
@LST()
```

Example:

```
DUP @CVI(@ABS(@LST()))      ; List unconditionally
```

LUN Convert Long Fractional to Floating Point

Converts a double-word long fractional to a floating-point value. The result's memory space attribute is N.

```
@LUN(expr)
```

Parameter

`expr`

A binary fraction expression.

Example:

```
DBLFRC EQU @LUN($3FE0000000000000)      ; DBLFRC = 0.5
```

MAC Macro Definition

Returns integer 1 if the specified symbol is defined as a macro name; otherwise returns 0. The result's memory space attribute is N.

```
@MAC(symbol)
```

Parameter

`symbol`

Any valid symbol.

Example:

```
IF  @MAC(DOMUL)       ; Expand macro
```

MAX Maximum Value

Determines which input expression has the greatest value, then returns that expression as a floating-point value. The result's memory space attribute is N.

`@MAX(expr1[,...,exprN])`

Parameters

`expr1 ... exprN`

Any valid expressions.

Example:

```
MAX DC @MAX(1.0,5.5,-3.25); MAX = 5.5
```

MIN Minimum Value

Determines which input expression has the least value, then returns that expression as a floating-point value. The result's memory space attribute is N.

`@MIN(expr1[,...,exprN])`

Parameters

`expr1 ... exprN`

Any valid expressions.

Example:

```
MIN DC @MIN(1.0,5.5,-3.25)       ; MIN = -3.25
```

MSP Memory Space

Returns the memory space attribute of the specified expression, as integer value 0 (for N) or 4 (for P).

`@MSP(expr)`

Parameter

`expr`

Any valid relative or absolute expression.

Example:

```
MEM SET @MSP(ORIGIN)      ; Save memory space
```

MXP Macro Expansion

Returns an integer 1 if the assembler is expanding a macro; otherwise returns a 0. The result's memory space attribute is N.

`@MXP()`

Example:

```
IF  @MXP()      ; Macro expansion active?
```

POS Position of Substring

Returns the position of substring `str2` in source string `str1` as an integer. Begins search at position `start`. If you omit the `start` value, the search begins at the beginning of `str1`. The result's memory space attribute is N.

`@POS(str1,str2[,start])`

Parameters

`str1`

Source string.

`str2`

Substring; must not exceed the length of `str1`.

`start`

A positive integer expression that does not exceed the length of `str1`.

Example:

```
ID EQU @POS('Star*Core 140','Core')      ; ID = 5
```

POW Raise to a Power

Returns the first expression, raised to the power of the second expression, as a floating-point value. A comma must separate the two expressions. The result's memory space attribute is N.

```
@POW(expr1,expr2)
```

Parameters

`expr1`

The expression whose value is to be raised.

`expr2`

The power-value expression.

Example:

```
BUF EQU @CVI(@POW(2.0,3.0))       ; BUF = 8
```

REL Relative Mode

Returns an integer 1 if the assembler is operating in relative mode; otherwise returns a 0. The result's memory space attribute is N.

```
@REL()
```

Example:

```
IF  @REL()      ; Check if in relative mode
```

RND Random Value

Returns a random value in the range 0.0 to 1.0. The result's memory space attribute is N.

```
@RND()
```

Example:

```
SEED DC @RND()      ; Save initial seed value
```

RVB Reverse Bits in Field

Reverses bits of the first expression, in the field the second expression delimits. Omitting the second expression makes the bit-reverse field the target word size.

```
@RVB(expr1[,expr2])
```

Parameters

`expr1, expr2`

Single-word, integer expressions.

Example:

```
REV EQU @RVB(VAL)      ; Reverse all bits in value
```

SCP Compare Strings

Returns an integer 1 if the specified strings are the same; otherwise returns 0. A comma must separate the two strings. The result's memory space attribute is N.

```
@SCP(str1,str2)
```

Parameters

`str1, str2`

String expressions.

Example:

```
IF  @SCP(STR,'MAIN')      ; Check if STR equals MAIN
```

SGN Return Sign

Returns the sign of the specified expression as an integer: -1 (negative), 0 (zero), or 1 (positive). The result's memory space attribute is N.

```
@SGN(expr)
```


For More Information: www.freescale.com

Parameter

`expr`

Any valid relative or absolute expression.

Example:

```
IF  @SGN(INPUT)      ; Check if sign is positive
```

SIN Sine

Returns the sine of the specified expression, as a floating-point value. The result's memory space attribute is N.

`@SIN(expr)`

Parameter

`expr`

Any valid expression.

Example:

```
DC  @SIN(@CVF(COUNT)*FREQ)      ; Compute sine value
```

SNH Hyperbolic Sine

Returns the hyperbolic sine of the specified expression, as a floating-point value. The result's memory space attribute is N.

`@SNH(expr)`

Parameter

`expr`

Any valid expression.

Example:

```
HSINE EQU @SNH(VAL)      ; Hyperbolic sine
```

SQT Square Root

Returns the square root of the specified expression, as a floating-point value. The result's memory space attribute is N.

```
@SQT(expr)
```

Parameter

`expr`

Any valid positive expression.

Example:

```
SQRT EQU @SQT(3.5)      ; SQRT = 1.870829
```

TAN Tangent

Returns the tangent of the specified expression, as a floating-point value. The result's memory space attribute is N.

```
@TAN(expr)
```

Parameter

`expr`

Any valid expression.

Example:

```
MOVE  #@TAN(1.0),D1.S      ; Load tangent
```

TNH Hyperbolic Tangent

Returns the hyperbolic tangent of the specified expression, as a floating-point value. The result's memory space attribute is N.

```
@TNH(expr)
```


For More Information: www.freescale.com

Parameter

`expr`

Any valid expression.

Example:

```
HTAN = @TNH(VAL)       ; Hyperbolic tangent
```

UNF Convert Fractional to Floating Point

Converts a fractional to a floating-point value. The result's memory space attribute is N.

`@UNF(expr)`

Parameter

`expr`

A binary fraction expression.

Example:

```
FRC EQU @UNF($400000)      ; FRC = 0.5
```

XPN Exponential Function

Returns the exponential function (base e raised to the power of the specified expression), as a floating-point value. The result's memory space attribute is N.

`@XPN(expr)`

Parameter

`expr`

Any valid expression.

Example:

```
EXP EQU @XPN(1.0)      ; EXP = 2.718282
```

For More Information: www.freescale.com

4

Software Project Management

Complex software projects often consist smaller program units. A team of programmers may write these subprograms in parallel, or they may reuse subprograms of a previous development effort.

This chapter explains the assembler directives that help manage complex software projects.

Sections

The SECTION and ENDSEC directives encapsulate program units. This defines relocatable blocks of code and data, postponing concerns about memory placement until after the assembly process.

A SECTION directive defines the start of a section, giving it the name that `section_name` specifies. The ENDSEC directive specifies the end of the section. The format is:

```
SECTION section_name [GLOBAL|STATIC|LOCAL] [core_id`]
.
.
source statements
.
.
ENDSEC
```

Section Names

Although you may give any name to a section, the assembler recognizes the names of conventional ELF sections — .text, .data, .rodata, and .bss. Table 4.1 lists the default types and flags for these section names. The assembler treats sections with other names as code (.text) sections, setting types and flags accordingly. If such a section is not a code section, you must use the SECTYPE and SECFLAGS directives to override the default settings.

Table 4.1 Conventional ELF Sections

Section	Contents	Type	Attributes
.bss	Uninitialized data	NOBITS	ALLOC, Write
.data	Initialized data	PROGBITS	ALLOC, WRITE
.mw_info	Assembler-generated contents that the linker consumes during dead data stripping	SHT_MW_INFO (SHT_LOPROC+3)	no sh_flags(0)
.note	User comments, as ABI 2.0 defines.	SHT_NOTE(7)	no sh_flags(0)
.rodata	Read-only, initialized data	PROGBITS	ALLOC
.text	Program code	PROGBITS	ALLOC, EXECINSTR

Table 4.2 lists the reserved names for specialized ELF sections; you should not used any of these names.

Table 4.2 Reserved Section Names

.debug_abbrev	.debug_pubname	.rel_line
.debug_aranges	.default	.rel.line.debug_info
.sc100.delay_slots	.hash	.rel.text
.debug_info	.line	.shstrtab
.debug_line	.mw_info	.strtab
.debug_loc	.note	.symtab
.debug_macinfo	.rel.sc100.delay_slots	
.debug_macro	.rel.debug_loc	

Nested and Fragmented Sections

You can nest sections to any level. When the assembler encounters a nested section, it stacks the current section and uses the new (child) section. When the assembler reaches the ENDSEC directive of the nested section, the assembler restores and resumes using the parent section. The ENDSEC directive always pertains to the most recent SECTION directive.You also can split sections into separate parts by using the same section name with multiple SECTION and ENDSEC directive pairs. Reusing a section name lets you

Freescale Semiconductor, Inc.
SC_Assem.book Page 77 Thursday, September 23, 2004 10:44 AM

arrange source statements arbitrarily, for example, grouping all statements that reserve P space storage locations.

Sections and Symbols

The default arrangement is that symbols defined within a section are local symbols. Any reference to a local symbol can be satisfied in the file in which it is defined.

Defining symbols outside a section makes them global. Such symbols can satisfy an outstanding current-file reference at assembly time, or an outstanding reference in any file at link time. Code inside or outside any section may reference global symbols freely, as long as the global symbol does not conflict with another symbol of the same name.

To declare a section's local symbols global:

- Use the GLOBAL directive for an individual symbol.
- Use the GLOBAL qualifier of the SECTION directive for all symbols in a section.

In Listing 4.1, `SYM1` and `SYM2` are global symbols, initially defined outside any section. But section `EXAMPLE` defines `SYM1` locally, with a different value:

- This interior redefinition means that the first `MOVE` instruction moves the value 3 to R0.
- `SYM2` remains a global symbol, so the second `MOVE` instruction moves the value 2 to R1.
- The final `MOVE` instruction is outside any section, so it uses the global `SYM1` definition, moving the value 1 to R2.

Listing 4.1 Sections and Data Hiding

```
SYM1    EQU 1
SYM2    EQU 2

        SECTION EXAMPLE

SYM1    EQU 3

        MOVE #SYM1,R0
        MOVE #SYM2,R1

        ENDSEC

        MOVE #SYM1,R2
```


For More Information: www.freescale.com

Macros and DEFINE Symbols within Sections

Macros and DEFINE directive symbols you define within a section are local. Global access never is possible for such macros and symbols.

To make macros or DEFINE symbols accessible globally, you must define them outside any section.

Sections and Relocation

Sections are the basic groups for relocating code and data blocks. With respect to relocation, code or data inside a section is an indivisible block, bound to a memory space. Within this memory space; such a code or data block is independently relocatable.

The assembler allocates a set of P-memory-space location counters for each section the source code defines. The assembler uses these counters to maintain data and instruction offsets from the beginning of the section. At link time, the linker can relocate sections to absolute addresses, load them in a particular order, or link them contiguously, as the programer specifies. If sections are split into parts or dispersed among files, the linker can recombine them logically, permitting relocation as a unit for each section.

Sections may be relocatable or absolute. If the assembler runs in absolute mode (command line `-a` option), all sections are absolute. If the assembler runs in relative mode (the default), all sections are relocatable initially.

To make a section or a part of a section absolute, use an ORG directive that specifies an absolute runtime address. If the assembler encounters such an ORG directive, it switches to absolute mode, generating absolute addresses. The assembler continues generating absolute code until it encounters an ENDSEC directive.

Address Assignment

The SC100 assembler supports either:

- Assigning absolute addresses at assembly time
- Generating relocatable program addresses to be resolved during linking

The assembler allows two sets of program counters: load counters and runtime counters. This distinction lets the assembler support *overlays* — runtime code/data transfers from one address to another. For example, you might load code and data at addresses A and B, but overlays could copy them to addresses F and N for execution.

Use the ORG directive to specify absolute address assignment. This directive also can specify the location counter (H, L, default, or a section's numbered counter), and assign initial values.

NOTE Counters 0, 1, and 2 correspond to the default, L, and H counters. Apart from this, there is no inherent relationship among numbered counters.
Location counter names default, L, and H are symbolic — the assembler does not verify that an H-counter value is greater than an L-counter value.

Counters are useful for providing mnemonic links among individual memory blocks. You can use separate counters for blocks in one section that get mapped to separate physical memories.

Overlays

When you use the SECTION directive, you define a regular section; you may use the SECFLAGS or SECTYPE directives to modify the section.

To define an overlay section, use the OVERLAY operand of the SECTYPE directive. Use the SECTYPE UNION directive to define a data overlay, as this example shows:

```
section .ovl_star
local                 section.ovl_pure_data
secflags nowrite,alloc,execinstr        secflags alloc
sectype overlay                             sectype union
```

Each overlay section has two starting addresses:

- **Load address —** address where the linker links the section.
- **Run address —** address where the section begins during execution of its code.

All symbol references symbols in an overlay section refer to the run address. To refer to a global symbol's load address, prefix the name with `LoadAddr_`, as Listing 4.2 shows. (You cannot make local symbols refer to load addresses.)

Listing 4.2 Referencing a Global Symbol's Load Address

```
      section .text local
      global _main
_main:
      push r0
      move.l #LoadAddr__star,r0
      jsr __overlay_manager
      . . .
      pop r0
```

```
      jsr _star
      rts

      section .ovl_text local
      secflags alloc,execinstr,nowrite
      sectype overlay
      global _star
_star:
      . . .
      rts
```

An overlay manager must copy an overlay section from its load address to its run address. In Listing 4.2, the code calls the overlay manager to load the overlay section `.ovl_text` to its run address. The overlay manager must know the overlay section's load address, which is unique (many overlays could run at the same address). After the overlay manager finishes, it is safe to call code in the overlay section.

You must provide a symbol at the start of each overlay section; the assembler does not create these symbols automatically.

Overlay Manager

Listing 4.3 shows a basic C implementation of an overlay manager:

Listing 4.3 Basic Overlay Manager

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct ovltab
 {
  void *ovl_run;
  void *ovl_load;
  unsigned long int ovl_size;
  unsigned long int ovl_checksum;
  unsigned long int ovl_flags;
  unsigned long int ovl_other;
  unsigned short int ovl_parent;
  unsigned short int ovl_sibling;
  unsigned short int ovl_child;
 }

extern struct ovltab _overlay_table[];
extern unsigned long int _overlay_count;

void *
```

```
_overlay_manager(void *load_addr)
 {
  unsigned long int i;

  for(i=0;i<_overlay_count; ++i){
      if(_overlay_table[i].ovl_load == load_addr){
        return memcpy(_overlay_table[i].ovl_run,
            _overlay_table[i].ovl_load,
            _overlay_table[i].ovl_size);
      }
  }

 return NULL;
}
```

Overlay Example

Listing 4.4 shows a sample assembly source file, `ovl.asm`, that defines overlay sections `ovl_star1` and `ovl_comet1`. To assemble this file, use the command:

```
asmsc100 -l -b ovl.asm
```

Listing 4.4 ovl.asm Assembly Source File

```
; main calling star1 and comet1

      section .text local
      global _main
_main:
      push r0
      move.l #LoadAddr__star1,r0
      jsr __overlay_manager
      pop r0
      jsr _star1

      push r0
      move.l #LoadAddr__comet1,r0
      jsr __overlay_manager
      pop r0
      jsr _comet1
      rts
      endsec

      section .ovl_star1 local
      secflags alloc,execinsr,nowrite
      sectype overlay
      global _star1
```

```
_star1:
      clr d1
      add #5,d1
      rts
      endsec

      section .ovl_comet1 local
      secflags alloc,execinsr,nowrite
      sectype overlay
      global _comet1
_comet1:
      clr d2
      add #5,d2
      rts
      endsec
```

The command

```
sc100-ld -Map ovl.map -c ovl.cmd ovl.eln
```

then links the resulting object file, using the linker command file that Listing 4.5 shows.

Listing 4.5 ovl.cmd Linker Command File

```
;*********************************************************************
;*
;* This memory control file defines the memory layout used with
;* the sc100 simulator. This file assumes small memory model
;* (i.e all globals and static data fit in lower 64k)
;*
;*    0          - 0x1ff            Interrupt vectors and handlers
;*  DataStart    - DataStart+DataSize-1      Global and static data
;*  CodeStart    - StackStart-1     Application code
;*  StackStart   - TopOfStack       stack/heap
;*  ROMStart     - TopOfMemory      ROM
;*
;*********************************************************************

;
; Define configuration specific values
;

.provide _DataStart,   0x0200        ; Start of global and static data
.provide _DataSize,  0x10000
.provide _CodeStart, 0x100000        ; Sets the code start address
.provide _StackStart,0x200000        ; Sets the stack start address
                                     ; The stack grows upwards.
.provide _TopOfStack, 0x2fff00       ; The highest address to be used
```

```
                                    ; by the C/C++ run-time.
                                    ; By default, this serves as the
                                    ; heap start address.
                                    ; The heap grows downwards.
.provide _ROMStart,    0x300000     ; Sets the ROM start address
.provide _TopOfMemory, 0x3fffff     ; The highest address in memory

.provide _SR_Setting, 0xe4000c      ; Value to set the SR after
reset:
                                    ;  exception mode
                                    ;  interrupt level 7
                                    ;  saturation on
                                    ;  rounding mode: nearest even

.memory 0, _TopOfMemory, "rwx"      ; Start execution at interrupt
                                    ; vector first entry (RESET).

.reserve _DataStart+_DataSize-2, _DataStart+_DataSize-1
                                    ; Reserve the top of the data
area
                                    ; to generate a linker error if
                                    ; data size does not fit 64k.
.reserve _StackStart, _TopOfStack   ; Reserve for stack and heap
space

.entry 0                            ; IntVec

.org 0

.overlay ".overlay1", "rwx", ".ovl_star1", ".ovl_star2", ".ovl_star3"
.overlay ".overlay2", "rwx", ".ovl_comet1", ".ovl_comet2"

.segment .intvec, ".intvec"

.org _DataStart
.segment .data,".data",".ramsp_0",".default",".bss"
.segment .ovltab, ".ovltab"

.org _CodeStart
.segment .text, ".text"
.segment .ovlstar, ".ovl_star*"
.segment .ovlcomet, ".ovl_comet*"
.segment .overlay1, ".overlay1"
.segment .overlay2, ".overlay2"

.org _ROMStart
.segment .rotable, ".init_table"
```

```
.segment .roinit, ".rom_init"
```

Multi-Programmer Environment Example

Developers often split multi-programmer projects into tasks that represent functional units. For example, suppose that a project has three task divisions: i/o, filter, and main. An individual programmer writes each task as a separate section:

- The I/O task yields file `io.asm` — Listing 4.6
- The filter task yields file `filter.asm` — Listing 4.7
- The main task yields file `main.asm` — Listing 4.8

Listing 4.6 io.asm Source File

```
          section i_o
          secflags alloc,write,noexecinstr
          global I_PORT
I_PORT
         .
         .
           source statements
         .
         .
          endsec
```

Listing 4.7 filter.asm Source File

```
          section filter
          secflags alloc,write,noexecinstr
         .
         .
           source statements
         .
         .
          endsec
```

Listing 4.8 main.asm Source File

```
          section main
          secflags alloc,write,noexecinstr
        global ENTRY
ENTRY
```

```
.
.
  source statements
.
.
 endsec
```

The next two subsections explain the methods for combining these three source files:

- Invoking a final assembly step to assign absolute addresses
- Assembling the modules separately, then linking.

Method 1: Absolute Mode

To assemble all project source code and assign absolute addresses, create a project file, `project.asm`, that combines the three assembly source files. Listing 4.9 shows this file.

Listing 4.9 project.asm File

```
org  PL:$1000              ; initialize entry point for program
counter

include 'main.asm'
include 'io.asm'
include 'filter.asm'
end  entry
```

To assemble file `project.asm` in absolute mode, and generate executable object file `project.eld`, use this command:

```
asmsc100 -a -bproject.eld project.asm
```

Upon reaching the end of file `main.asm`, the assembler returns to the next sequential statement in file `project.asm` . This statement directs the assembler to start taking input from file `io.asm`. The ORG statement in the `project.asm` file tells the assembler to set current memory space to P (program), and to initialize the L (low) location counter to $1000. This specifies assembly of the `io.asm` statements at the next available Low Program memory space.

Assembly of file `filter.asm` happens in a similar manner. The last line of file `project.asm` tells the assembler that it is the last logical source statement. The last line also tells the assembler that the starting address for the object module is label ENTRY. (In actual code, the ENTRY label must be declared global in section `main`.)

Method 2: Relative Mode

Use the assembler default mode to assemble each source file separately. Use these commands:

```
asmsc100 -bmain.eln main.asm
asmsc100 -bio.eln io.asm
asmsc100 -bfilter.eln filter.asm
```

In response, the assembler generates relocatable object files `main.eln`, `io.eln`, and `filter.eln`. The assembler establishes a separate set of location counters for each input-file section. This means that all memory spaces for each section begin at relative address zero.

Use this command to invoke the linker:

```
sc100-ld -M -c link.cmd -o project.eld main.eln io.eln
filter.eln
```

In response, the linker combines the relocatable object files, reading each section's address and the program entry point from linker command file `link.cmd`. The linker processes the three `.eln` files in their command-line order, outputting executable file `project.eld`.

5

Assembler Directives

This chapter explains the special characters significant to the assembler, as well as the directives that control assembler behavior. In response to these directives, the assembler carries out certain actions during assembly. But it is not appropriate for the assembler to translate every directive into machine language.

Significant Characters

Several one- and two-character sequences have significance for the assembler, their meanings sometimes depending on context. Table 5.1 identifies these characters briefly; full explanations appear at the start of this chapter's reference items.

(Chapter 3 explains special characters for expression evaluation.)

Table 5.1 Assembler Significant Characters

Character	Description
;	Comment delimiter
;;	Unreported comment delimiter
\	Line continuation character; macro dummy argument concatenation operator
?	Macro value substitution operator
%	Macro hexadecimal value substitution operator
^	Macro local label override operator
“	Macro string delimiter; quoted string DEFINE expansion character
@	Function delimiter
*	Location counter substitution
++	String concatenation operator
[ ]	Substring delimiters; instruction grouping operators
<	Short addressing mode force operator

For More Information: www.freescale.com

Table 5.1 Assembler Significant Characters (*continued*)

Character	Description
>	Long addressing mode force operator
#	Immediate addressing mode operator
#<	Immediate short addressing mode force operator
#>	Immediate long addressing mode force operator
$	Hexadecimal constants indicator
‘	String constants delimiter

Directive List

Table 5.2 briefly identifies the assembler directives of each type:

- Assembly control
- Symbol definition
- Data definition/storage allocation
- Listing control and options
- Macros and conditional assembly

Table 5.2 Assembler Directive Summary

Type	Directive	Description
Assembly Control	COMMENT	Start comment lines
	DEFINE	Define substitution string
	END	End of source program
	FAIL	Programmer-generated error message
	HIMEM	Set high memory bounds
	INCLUDE	Include secondary file
	LOMEM	Set low memory bounds
	MSG	Programmer-generated message
	ORG	Initialize memory space and location counters
	RADIX	Change input radix for constants
	UNDEF	Undefine the DEFINE symbol
	WARN	Programmer-generated warning
Symbol Definition	ENDSEC	End section
	EQU	Equate symbol to a value
	GLOBAL	Global section symbol declaration
	GSET	Set global symbol to a value
	SECFLAGS	Set ELF section flags
	SECTION	Start section
	SECTYPE	Set ELF section type
	SET	Set symbol to a value
	SIZE	Set size of symbol in the ELF symbol table
	TYPE	Set symbol type in the ELF symbol table
	XDEF	Define external symbol (COFF only)
	XREF	Reference external symbol (COFF only)

Table 5.2 Assembler Directive Summary (*continued*)

Type	Directive	Description
Data Definition and Storage Allocation	ALIGN	Align location counter
	BADDR	Set buffer address
	BSB	Block storage bit-reverse
	BSC	Block storage of constant
	BSM	Allocate storage block for modulo (COFF only)
	BUFFER	Start buffer
	DC	Define constant
	DCB	Define constant byte
	DCL	Define constant long
	DCW	Define constant word
	DS	Define storage
	DSM	Define storage for modulo (COFF only)
	DSR	Define reverse carry storage
	ENDBUF	End buffer
	FALIGN	Fetch-set alignment
Listing Control and Options	LIST	List the assembly
	NOLIST	Stop assembly listing
	NOTE	Include note
	OPT	Set assembler options
	PAGE	Top of page/size page
	PRCTL	Send control string to printer
	STITLE	Initialize program subtitle
	TITLE	Initialize program title

For More Information: www.freescale.com

Table 5.2 Assembler Directive Summary (*continued*)

Type	Directive	Description
Macros and Conditional Assembly	DUP	Duplicate sequence of source lines
	DUPA	Duplicate sequence with arguments
	DUPC	Duplicate sequence with characters
	DUPF	Duplicate sequence in loop
	ENDIF	End of conditional assembly
	ENDM	End of macro definition
	EXITM	Exit macro
	IF	Conditional assembly directive
	MACLIB	Macro library
	MACRO	Macro definition
	PMARCRO	Purge macro definition

Descriptions

The rest of this chapter consists of special-character and directive descriptions, including usage guidelines and examples:

- Descriptions of special characters follow the order of Table 5.1. (The $ and ‘, hexadecimal and string delimiter characters, however, are so simple that they do not require such descriptions.
- Descriptions of directives are in alphabetic order, without regard the type categories of Table 5.2.

NOTE This chapter shows directives in upper-case letters, but the assembler recognizes either case for directives.
You may not use a label on the same line as a directive, unless the prototype includes a label parameter.

; Start Comment

Starts a comment: any number of characters not part of a literal string.

Remarks

For a comment in a source-statement line, shift the ; character right, so that the comment lines up with comments of other lines. For a comment that takes up an entire line, put the ; character at the first space of the line.

Use comments to document your source program: although the assembler reproduces comments in the source listing, comments are not significant to the assembler. Macro definitions preserve comments, but you can use the NOCM option to turn off this arrangement.

Example

```
; This comment begins in column 1 of the source file
LOOP    JSR COMPUTE      ; This is a trailing comment
                         ; a source-file tab precedes
                            ; these two comments
```

;; Start Unreported Comment

Starts an unreported comment: any comment that the assembler does not reproduce in the source listing, nor save in macro definitions.

Remarks

Unreported comments follow the same position rules as normal comments; you can use them to document your source program. However, such comments never appear in the assembler source listing, nor does the assembler save them in macro definitions.

Example

```
;; These lines will not be reproduced
;; in the source listing
```

\ Continue Line

Continues a source statement to the next line.

Remarks

If the assembler encounters a backslash (\) as the last character of a source line, it concatenates the line to the next line, processing the result as if it were a statement on one line. The maximum length for such a concatenated line is 4000 characters.

(Alternate role: In a macro definition, this character concatenates a dummy argument with adjacent characters.)

Example

The \ character makes one comment span three lines:

```
; This comment \
extends over \
three lines.
```

\ Concatenate Macro Argument

In a macro definition, concatenates a dummy argument with adjacent characters.

Remarks

There must not be any spaces with the \ character, which can precede or follow the adjoining characters. To position an argument between two characters, use the \ character before and after the argument name.

(Alternate role: If the last character of a source line, continues the statement to the next line.)

Example

The backslash (\) in the macro definition tells the macro processor to concatenate the dummy-argument substitution characters with the character R:

```
SWAP_REG  MACRO REG1,REG2     ; Swap REG1,REG2 (D4 is
temp)
          MOVE  R\REG1,D4
          MOVE  R\REG2,R\REG1
          MOVE  D4,R\REG2
          ENDM
```

The macro call `SWAP_REG 0,1` results in this expansion:

```
MOVE R0,D4
MOVE R1,R0
MOVE D4,R1
```

? Substitute Macro Value

In macro definitions, converts a `symbol` to the ASCII string that represents the symbol's decimal value. There must not be any spaces between the ? character and the `symbol`; the value of `symbol` must be an integer. You may use the ? character with the backslash (\) concatenation operator.

Example

Consider this macro definition:

```
SWAP_SYM  MACRO REG1,REG2       ; Swap REG1,REG2 (D4 is
temp)
          MOVE  R\?REG1,D4
          MOVE  R\?REG2,R\?REG1
          MOVE  D4,R\?REG2
           ENDM
```

Then suppose these SET statements and this macro call:

```
AREG      SET 0
BREG      SET 1
           SWAP_SYM AREG,BREG
```

The macro processor would:

- Substitute the characters `AREG` for each occurrence of `REG1`, and `BREG` for each occurrence of `REG2`, as if producing this intermediate macro expansion:

```
MOVE R\?AREG,D4
MOVE R\?BREG,R\?AREG
MOVE D4,R\?BREG
```

- Replace `?AREG` with the character `0`, and `?BREG` with the character `1`, as if producing this second intermediate expansion:

```
MOVE R\0,D4
MOVE R\1,R\0
MOVE D4,R\1
```

- Apply the concatenation operator (\), producing the expansion that appears in the source listing:

```
MOVE R0,D4
```

```
MOVE R1,R0
MOVE D4,R1
```

% Substitute Macro Hex Value

In macro definitions, converts a `symbol` to the ASCII string that represents the symbol's hexadecimal value.

Remarks

There must not be any spaces between the % character and the `symbol`; the value of `symbol` must be an integer. You may use the % character with the backslash (\) concatenation operator.

The % character also can indicate a binary constant. If you need a binary constant inside a macro, enclose the constant in parentheses. Alternatively, follow the percent sign with a backslash (\) to escape the constant.

Example

This macro definition generates a label — the label prefix argument concatenated to a hexadecimal argument:

```
GEN_LAB    MACRO  LAB,VAL,STMT
LAB\%VAL   STMT
           ENDM
```

Suppose this SET statement and this macro call:

```
NUM       SET  10
           GEN_LAB HEX,NUM,'NOP'
```

The macro processor would:

- Substitute the characters HEX for LAB.
- Replace %VAL with the character A (the hexadecimal equivalent of decimal 10).
- Apply the concatenation operator (\).
- Substitute the string 'NOP' for the STMT argument, producing this expansion:

```
HEXA       NOP
```

^ Override Macro Local Label

If a unary expression operator in a macro expansion, specifies normal-scope (not macro-scope) evaluation for local labels in its associated term.

Remarks

If the circumflex (^) character precedes an expression term, the assembler does not search the macro local label list for any %labels in the expression term. This operator has no effect on normal labels; it has no effect at all outside a macro expansion.

The circumflex operator lets you pass local labels as macro arguments, for use as referents in the macro. Note that the circumflex also is the binary exclusive OR operator.

Example

Consider this macro definition:

```
LOAD        MACRO ADDR
            MOVE  P:^ADDR,R0
              ENDM
```

And this macro call:

```
%LOCAL
          LOAD %LOCAL
```

The override operator tells the assembler to recognize the %LOCAL symbol outside the macro expansion, and to use that value in the MOVE instruction. If the override operator were not present, the assembler would issue an error message that %LOCAL was not defined in the macro.

" Delimit Macro String

In a macro definition, tells the macro processor to use a single quote (**'**). This transforms any enclosed dummy arguments into literal strings.

(Alternate role: In a DEFINE-directive character sequence, specifies expansion within the string.)

Example

For this macro definition:

```
CSTR        MACRO STRING
            DC "STRING"
             ENDM
```

And this macro call:

```
             CSTR ABCD
```

The macro expansion would be:

```
             DC 'ABCD'
```

" Expand DEFINE Quoted String

In a DEFINE-directive character sequence, specifies expansion within the string. (Otherwise, delimits a string, just as single quotes.)

(Alternate role: In a macro definition, tells the macro processor to use a single quote.)

Example

For this macro definition:

```
            DEFINE LONG 'short'
STR_MAC     MACRO STRING
            MSG 'This is a LONG STRING'
            MSG "This is a LONG STRING"
             ENDM
```

and this macro call:

```
             STR_MAC sentence
```

the macro expansion would be:

```
MSG  'This is a LONG STRING'
MSG  'This is a short sentence'
```

@ Start Function

Mandatory start symbol for all assembler built-in functions.

Example

```
SVAL    EQU @SQT(FVAL)        ; Obtain square root
```

* Substitute Location Counter

If an operand in an expression, represents the current integer value of the location counter.

Example

```
          ORG P:$100
PBASE     EQU *+$20         ; PBASE = $120
```

++ Concatenate Strings

Concatenates any two strings. Single or double quotes must enclose the strings; there must not be intervening spaces.

Example

```
'ABC'++'DEF' = 'ABCDEF'
```

[ ] Delimit Substring

Delimit a substring operation. (Alternate role: Group instructions.)

`[string,offset,length]`

Parameters

`string`

Source string: any valid string combination, including another substring.

`offset`

Substring starting position within the source string, beginning at 0; may not exceed the length of the source string.

`length`

Length of the substring; may not exceed the length of the source string.

Example

```
DEFINE ID ['abcdefg',1,3]        ;ID = 'bcd'
```

[ ] Group Instructions

Group instructions. The opening bracket cannot appear in the label field. If the first instruction is on the same line, a space must separate the bracket and the instruction. (Alternate role: Delimit substrings.)

```
[
instruction ...
]
```

Example

```
[
  MAC D0,D1,D2     MAC D3,D4,D5       ; multiply
operands
  ADD D0,D1,D3     ADD D3,D4,D6       ; add operands
  MOVE.F (R0)+,D0  MOVE.W (R1)+,D1    ; load new
operands
]
```

< Force Short Addressing

Forces the assembler to use short absolute addressing, overriding default long addressing.

Remarks

Many DSP instructions permit a short form of addressing. If the assembler knows the absolute address during pass one, the assembler uses the shortest addressing mode consistent with the instruction format.

But if an address is a forward or external reference, the assembler cannot know the absolute address during pass one. Accordingly, the assembler uses the long form of addressing; this makes the instruction two words. To override this default arrangement, start the absolute address with the < character.

Example

In this sample code:

```
MOVE.B D0,P:DATAST
```

```
DATAST    EQU    P:$23
```

The DATAST symbol is a forward reference; the assembler uses long absolute addressing: two words. To force short absolute addressing, insert the < character:

```
          MOVE.B D0,P:<DATAST
DATAST    EQU    P:$23
```

> Force Long Addressing

Forces the assembler to use long absolute addressing.

Remarks

Many DSP instructions permit a long form of addressing. But if the assembler knows the absolute address during pass one, the assembler uses the shortest addressing mode consistent with the instruction format. To override this behavior, forcing long absolute addressing, start the absolute address with the > character.

Example

In this sample code:

```
DATAST    EQU     P:$23
           MOVE.B D0,P:DATAST
```

The DATAST symbol is *not* a forward reference; the assembler uses short absolute addressing. To force long absolute addressing, insert the > character:

```
DATAST    EQU     P:$23
           MOVE.B D0,P:>DATAST
```

Use Immediate Addressing

Tells the assembler to use immediate addressing mode.

Example

```
CNST     EQU  $5
          MOVE.B #CNST,D0
```

Freescale Semiconductor, Inc.
SC_Assem.book Page 101 Thursday, September 23, 2004 10:44 AM

#< Force Immediate Short Addressing

Forces the assembler to use immediate short addressing, overriding default immediate long addressing.

Remarks

Many DSP instructions permit a short immediate form of addressing. If the assembler knows the immediate data during pass one, the assembler uses the shortest addressing mode consistent with the instruction format.

But if the immediate data is a forward or external reference, the assembler cannot know the immediate data during pass one. Accordingly, the assembler uses the long form of immediate addressing; this makes the instruction two words. To override this default behavior, start the immediate data symbol with the #< characters.

Example

For this sample code:

```
        MOVE.B #CNST,D0
CNST     EQU  $5
```

The assembler does not know the `CNST` symbol during pass one; the assembler uses immediate long addressing: two words. To force immediate short addressing, insert the #< characters:

```
        MOVE.B #<CNST,D0
CNST     EQU  $5
```

#> Force Immediate Long Addressing

Forces the assembler to use immediate long addressing.

Remarks

Many DSP instructions permit a long immediate form of addressing. But if the assembler knows the immediate data during pass one, the assembler uses the shortest addressing mode consistent with the instruction format. To override this behavior, forcing immediate long addressing, start the immediate data symbol with the #> characters.

For More Information: www.freescale.com

Example

For this code:

```
CNST    EQU $5
         MOVE.B #CNST,D0
```

The DATAST symbol is *not* a forward reference; the assembler uses short absolute addressing. To force long absolute addressing, insert the > character:

The assembler knows the CNST symbol during pass one; the assembler uses immediate short addressing. To force immediate long addressing, insert the #> characters:

```
CNST    EQU $5
         MOVE.B #>CNST,D0
```

ALIGN Align Location Counter

Advances the location counter, aligning it on the specified address boundary. If the location counter already is aligned on this boundary, this directive has no effect.

```
ALIGN boundary
```

Parameter

`boundary`

Address boundary specifier; must be a power of two.

Example

```
ALIGN 4         ; Align location counter to
                 ; next long word boundary
```

BADDR Set Buffer Address

Sets the location counter to the address of a reverse-carry buffer. Does not initialize the block of memory intended for the buffer.

```
BADDR  R,length
```

Parameters

`R`

Specifier for the reverse carry buffer type.

`length`

Buffer length, in bytes: an expression that evaluates to an absolute integer greater than zero. Must not contain any forward references; should be a power of two.

Remarks

If the location counter value is not zero, this directive advances the location counter to a base address that is a multiple of 2^k, where:

$2^k \geq$ `length`

There must be sufficient remaining memory to establish a valid base address, or the assembler issues an error message. Unlike other buffer allocation directives, the BADDR directive does *not* advance the location counter; the location counter continues to point to the buffer base address.

You may not use a label with this directive. The assembler issues a warning if the `length` value is not a power of two.

Related Directives

BSB, BUFFER, DSR

Example

```
ORG P:$100
BADDR R,24        ; Reverse buffer 24
```

BSB Allocate Bit-Reverse Buffer

Allocates and initializes a block of bytes for a reverse-carry buffer.

```
[label]  BSB  length[,value]
```

Parameters

`label`

Optional label that receives the value of the location counter, once the assembler establishes a valid base address.

`R`

Specifier for the reverse carry buffer type.

`length`

Block length, in bytes: an expression that evaluates to an absolute integer greater than zero. Must not contain any forward references; should be a power of two. Can have any memory space attribute.

`value`

Optional value expression for the initial value of each array byte. Can have any memory space attribute. Omitting this value tells the assembler to use the value zero.

Remarks

If the location counter is not zero, this directive advances the location counter to a base address that is a multiple of 2^k, where:

$2^k \geq$ `length`

The listing shows only one byte of object code, regardless of how large the `length` expression is. However, the location counter advances by the number of bytes generated.

The assembler issues a warning if the `length` expression is not a power of two.

Related Directives

BADDR, BSC, DC

Example

```
BUFFER  BSB  BUFSIZ        ; Initialize buffer to zeros
```

BSC Allocate Constant Storage Block

Allocates and initializes a block of bytes.

```
[label]  BSC length[,value]
```

Parameters

`label`

Optional label that receives the value of the location counter at the start of directive processing.

`length`

Block length, in bytes: an expression that evaluates to an absolute integer greater than zero. Must not contain any forward references. Can have any memory space attribute.

`value`

Optional value expression for the initial value of each block byte, in the range -128..+255. Can have any memory space attribute. Omitting this value tells the assembler to use the value zero.

Remarks

The listing shows only one byte of object code, regardless of how large the `length` expression is. However, the location counter advances by the number of bytes generated.

Related Directives

BADDR, BSB, DC

Example

```
UNUSED  BSC  $2FFF-@LCV(R),$FFFFFFFF   ; Fill unused
EPROM
```

BSM Allocate Block Storage for Modulo

For COFF mode only, allocates and initializes a block of words for a modulo buffer.

```
[label]  BSM numwords[,value]
```

Parameters

`label`

Optional label that receives the value of the location counter after establishment of a valid base address.

`numwords`

Block length, in words: an expression that evaluates to an absolute integer greater than zero. Must not contain any forward references. Can have any memory space attribute.

`value`

Optional value expression for the initial value of each block word. Can have any memory space attribute. Omitting this value tells the assembler to use the value zero.

Remarks

This directive pertains to COFF mode only; ELF mode does not support this directive.

If the value of the runtime location counter is not zero, this directive advances the runtime location counter to a base address that is a multiple of 2^k, where 2^k equals or exceeds the `numwords` value. The assembler issues an error message if:

- `numwords` contains symbols not yet defined,

- the `numwords` value is zero or less, or
- the `numwords` value exceeds the target-DSP maximum address.

Related Directives

BSB, BSC, DC

Example

```
BUFFER  BSM  BUFSIZE,$FFFFFFFF   ; Initialize buffer to
1
```

BUFFER Start Buffer

Indicates the beginning address of a reverse-carry buffer; does not initialize .

```
BUFFER R,length
```

Parameters

`R`

Specifier for the reverse carry buffer type.

`length`

Buffer length, in bytes: an expression that evaluates to an absolute integer greater than zero. Must not contain any forward references; should be a power of two.

Remarks

In response to this directive, the assembler allocates data for the buffer until it encounters an ENDBUF directive. If allocated data does not fill the buffer, unfilled locations remain uninitialized; if allocated data exceeds the specified buffer size, the assembler issues an error message.

Instructions and most data definition directives may appear between the BUFFER and ENDBUF directive pair. But you must not nest BUFFER directives. The directives that may not appear between BUFFER and ENDBUF are:

- MODE
- ORG
- SECTION
- Other buffer allocation directives

The BUFFER directive sets the location counter to the address of a buffer of the given type. If the location counter is not zero, this directive advances the location counter to a base address that is a multiple of 2^k, where:

$2^k \geq$ `length`

There must be sufficient remaining memory to establish a valid base address, or the assembler issues an error message. Unlike other buffer allocation directives, the BUFFER directive does *not* advance the location counter; the location counter continues to point to the buffer base address.

You may not use a label with this directive. The assembler issues a warning if the `length` value is not a power of two.

Related Directives

BADDR, BSB, DSR, ENDBUF

Example

```
          ORG P:$100
          BUFFER R,28              ; Reverse buffer 28
R_BUF     DC 0.5,0.5,0.5,0.5
          DS 20                    ; Remainder
uninitialized
           ENDBUF
```

COMMENT Start Comment Lines

Defines one or more lines as comments.

```
COMMENT delimiter
     .
     .
     delimiter
```

Parameter

`delimiter`

Any non-space character.

Remarks

The two delimiter characters define comment text. The line that contains the second delimiter character is the last line of the comment. Comment text can include any printable characters; the assembler does reproduce this text in the source listing.

You may not use a label with this directive.

Examples

```
COMMENT  +  This is a one-line comment +
COMMENT  *  This is a multiple-line
            comment. Any number of lines
            can be placed between the two delimiters.
         *
```

DC Define Constant

Allocates and initializes two bytes of memory for each argument.

```
[label]DC  arg[,arg,...]
```

Parameters

`label`

Optional label that receives the value of the location counter at the start of directive processing.

`arg`

An integer constant, fractional constant, symbol, or expression. Commas without spaces must separate multiple `arg` values.

Remarks

The assembler stores multiple arguments in successive address locations. If this directive has multiple arguments, one or more can be null (two adjacent commas): this fills the corresponding address location with zeros.

If you use the DC directive in L memory, the assembler evaluates and stores arguments as long word quantities. Otherwise, the assembler issues an error message if the evaluated argument value is too large for a single DSP word.

The assembler stores integer arguments as integers; it converts floating-point arguments to binary values. String storage is:

- Single-character strings: a word whose lower seven bits represent the ASCII character value. For example, the assembler stores the string 'R' as $000052.
- Multiple-character strings: words that are concatenated sequences of ASCII values. If the number of string characters is not an even multiple of the number of bytes per DSP word, the last word's remaining characters are left aligned, and zeros fill the rest of the word. For example, the assembler stores the string 'ABCD' as two words, $414243 and $410000.

- Exception: The NOPS option tells the assembler to store each character of a string as it would store a single-character string. It would store the string ‘ABCD’ as $000041, $000042, $000043, and $000044.

Related Directives

DCB, DCL

```
TABLE    DC 1426,253,$2662,'ABCD'
CHARS     DC 'A','B','C','D'
```

DCB Define Constant Byte

Allocates and initializes a byte of memory for each argument.

```
[label]  DCB arg[,arg,...]
```

Parameters

`label`

Optional label that receives the value of the location counter at the start of directive processing.

`arg`

A byte integer constant, string constant, symbol, or byte expression. Integer constants must be byte values, in the range 0 — 255. Commas without spaces must separate multiple `arg` values.

Remarks

The assembler stores multiple arguments in successive byte locations. If this directive has multiple arguments, one or more can be null (two adjacent commas): this fills the corresponding byte location with zeros.

The assembler stores integer arguments as integers; you may not use floating-point arguments. String storage is:

- Single-character strings: a byte whose lower seven bits represent the ASCII character value. For example, the assembler stores the string 'R' as $52.
- Multiple-character strings: consecutive bytes, each of which contains an ASCII value. For example, the assembler stores the arguments ‘AB’,,’CD’ as $41, $42, $00, $43, and $44.

Related Directives

DC, DCL

Example

```
TABLE    DCB 'two',0,'strings',0
CHARS     DCB 'A','B','C','D'
```

DCL Define Constant Long

Allocates and initializes four bytes of memory for each argument.

```
[label]DCL arg[,arg,...]
```

Parameters

`label`

Optional label that receives the value of the runtime location counter at the start of directive processing.

`arg`

An integer constant, fractional constant, symbol, or expression. Commas without spaces must separate multiple `arg` values.

Related Directives

DC, DCB

Example

```
DCL $12345678      ; Assuming little endian mode,
                   ; $12345678 = $78
                   ;             $56
                   ;             $34
                     ;              $12
```

DEFINE Define Substitution String

Defines substitution strings that the assembler uses in all following source lines.

```
DEFINE symbol string
```

Parameters

`symbol`

Valid global or local symbol that appears in source lines.

`string`

Replacement string for the symbol.

Remarks

Upon encountering a macro definition, the assembler applies DEFINE-directive translations. Later, when the assembler expands the macro, it applies the appropriate DEFINE-directive translations to the expansion.

You may not use a label with this directive.

Related Directives

GSET, SET, UNDEF

Example

```
          DEFINE ARRAYSIZ '16*SAMPLSIZ'
SAMPLSIZ  EQU    16
          DS     ARRAYSIZ    ; This line transformed to
          .                  ; DS 16*SAMPLSIZ
          .
```

DS Define Storage

Reserves a block of bytes in memory, but does not initialize the block.

```
[label]  DS numbytes
```

Parameters

`label`

Optional label that receives the value of the location counter at the start of directive processing.

`numbytes`

Number of bytes: an expression that evaluates to an integer greater than zero. This expression must not contain any forward references. The location counter advances by this number of bytes.

Related Directive

DSR

Example

```
; If the current loader address is $9e
      align16; Align on next 16-byte boundary
R_BUF DS    8    ; Reserve 8 bytes for R_BUF
S_BUF DS     12    ; Reserve 12 bytes for S_BUF
```

DSM Define Storage for Modulo

For COFF mode only, reserves a block of words for modulo storage; does not initialize the word values.

```
[label]  DSM numwords
```

Parameters

`label`

Optional label that receives the value of the location counter after establishment of a valid base address.

`numwords`

Block length, in words: an expression that evaluates to an absolute integer greater than zero. Must not contain any forward references. Can have any memory space attribute.

Remarks

This directive pertains to COFF mode only; ELF mode does not support this directive.

If the value of the runtime location counter is not zero, this directive advances the runtime location counter to a base address that is a multiple of 2^k, where 2^k equals or exceeds the `numwords` value. The assembler issues an error message if:

- `numwords` contains symbols not yet defined,
- the `numwords` value is zero or less,
- the `numwords` value exceeds the target-DSP maximum address, or
- there is insufficient memory to establish a valid base address.

Related Directives

DS, DSR

Example

```
        ORG  P:$100
M_BUF   DSM  24         ; Circular buffer mod 24
```

DSR Define Reverse-Carry Storage

Reserves a block of bytes in memory for a reverse-carry buffer, but does not initialize the block.

```
[label]  DSR numbytes
```

Parameters

`label`

Optional label that receives the value of the location counter, once the assembler establishes a valid base address.

`numbytes`

Number of bytes: an expression that evaluates to an absolute integer greater than zero. This expression should be a power of two, and must not contain any forward references.

Remarks

This directive advances the location counter from a valid base address:

- If the location-counter value is not zero, the assembler advances the location counter to a base address that is a multiple of 2,^{k} where $2^k \geq$ `numbytes`. The assembler issues an error message if there is insufficient memory.
- Then the assembler advances the location counter by the value of `numbytes`.

The assembler generates a warning if `numbytes` is not a power of two.

Related Directive

DS

Example

```
      ORG P:$100 ; Set address to P:$100
R_BUF DSR 8      ; Reserve 8 bytes for R_BUF
```

DUP Duplicate Source Lines

Duplicates the following source lines the specified number of times; the ENDM directive marks the last line to be duplicated.

```
[label]  DUP times
                .
                .
                 ENDM
```

Parameters

`label`

Optional label that receives the value of the runtime location counter at the start of directive processing.

`times`

Number of times to duplicate source lines: an expression that evaluates to an absolute integer. This expression can have any memory-space attribute, but must not contain any forward references. If the times value is less than or equal to zero, the assembler output does not include the sequence of lines.

Remarks

You may nest the DUP directive to any level. If the `times` value is less than or equal to zero, the assembler output does not include the sequence of lines.

To immediately halt source-line duplication, for example, upon detection of an error, use the EXITM directive and conditional-assembly directives.

Related Directives

DUPA, DUPC, DUPF, DUPA, EXITM, MACRO

Example

If MD and MEX options are enabled, and if input includes these lines:

```
COUNT SET   3
      DUP   COUNT     ; ASR BY COUNT
      ASR   D0
      ENDM
```

The source listing includes the line `ASR   D0` three times.

DUPA Duplicate Sequence with Arguments

Duplicates the following source statements for each argument, substituting successive `arg` values for the `dummy` argument. The ENDM directive marks the last statement to be duplicated. To immediately halt source-line duplication, for example, upon detection of an error, use the EXITM directive and conditional-assembly directives.

```
[label]  DUPA  dummy,arg[,arg,...]
                .
                .
                 ENDM
```

Parameters

`label`

Optional label that receives the value of the runtime location counter at the start of directive processing.

`dummy`

Valid expression that appears in the source lines.

`arg`

Argument string. Single quotes must enclose an embedded space or other character significant to the assembler. If a null string, the assembler removes `dummy` values as it repeats the statements. Commas without spaces must separate multiple `arg` values.

Related Directives

DUP, DUPC, DUPF, ENDM, EXITM, MACRO

Example

If MD and MEX options are enabled, and if input includes these lines:

```
DUPA VALUE,12,32,34
DC    VALUE
 ENDM
```

The source listing shows successive lines:

```
DC   12
DC   32
 DC   34
```

DUPC Duplicate Sequence with Characters

Duplicates the following source statements for each character of the string argument, substituting successive `string` characters for the `dummy` argument. The ENDM directive marks the last statement to be duplicated. To immediately halt source-line duplication, for example, upon detection of an error, use the EXITM directive and conditional-assembly directives.

```
[label]  DUPC  dummy,string
                .
                .
                 ENDM
```

Parameters

`label`

Optional label that receives the value of the runtime location counter at the start of directive processing.

`dummy`

Valid expression that appears in the source lines.

`String`

Valid string expression.Argument string. Single quotes must enclose an embedded space or other character significant to the assembler. If a null string, the assembler skips the block of statements.

Related Directives

DUP, DUPA, DUPF, ENDM, EXITM, MACRO

Example

If MD and MEX options are enabled, and if input includes these lines:

```
DUPC VALUE,'123'
DC   VALUE
 ENDM
```

The source listing shows successive lines:

```
DC  1
DC  2
 DC  3
```

DUPF Duplicate Sequence in Loop

Duplicates the following source statements; the values of `start`, `end`, and `increment` arguments determine the number of duplications. The ENDM directive marks the last statement to be duplicated. To immediately halt source-line duplication, for example, upon detection of an error, use the EXITM directive and conditional-assembly directives.

```
[label]  DUPF  dummy[,start],end[,increment]
                .
                .
                ENDM
```

Parameters

`label`

Optional label that receives the value of the runtime location counter at the start of directive processing.

`dummy`

A parameter that holds the loop index value; may appear in instruction statements.

`start`

Optional starting value for the loop index; defaults to 1.

`end`

Ending value for the loop index.

`increment`

Optional increment value for the loop index; defaults to 1.

Related Directives

DUP, DUPA, DUPC, ENDM, EXITM, MACRO

Example

If MD and MEX options are enabled, and if input includes these lines:

```
DUPF    NUM,0,7
MOVE.B #0,R\NUM
ENDM
```

The source listing includes eight copies of the line `MOVE.B #0,R\NUM`.

For More Information: www.freescale.com

ELSE Start Alternative Conditional Assembly

Delimits alternative conditional assembly: ends or begins source lines to be assembled, according to the value of the assembly condition.

```
ELSE
```

Remarks

An optional directive valid only with a pair of IF and ENDIF directives:

- If the IF directive's `condition` value is TRUE (non-zero), the assembler assembles source lines between the IF and ELSE directives, ignores lines between the ELSE and ENDIF directives.
- If the IF directive's `condition` value is FALSE (zero), the assembler ignores source lines between the IF and ELSE directives, assembles lines between the ELSE and ENDIF directives.

You can nest conditional directives to any level. The ELSE directive, like the ENDIF directive, always pairs with the closest previous IF directive. You may not use a label with this directive.

Related Directives

ENDIF, IF

Example

If the mode is little-endian and the assembler encounters this code:

```
IF @DEF('BIG_ENDIAN')
  MOVE.W  (r0)-,d1        ; Start traceback from
ELSE                      ;   state zero.
  MOVE.W  (r0),d1
ENDIF
```

The assembler ignores the line `MOVE.W  (r0)-,d1` and assembles the line `MOVE.W  (r0),d1`.

END End of Source Program

Marks that the logical end of the source program; the assembler ignores any statements following this directive.

```
END  [startaddr]
```

Parameter

`startaddr`

Optional starting execution address of the program. Only valid for absolute mode; must have memory-space attribute P (program) or N (none).

Remarks

You cannot use this directive in a macro expansion. You may not use a label with this directive.

Example

```
END  BEGIN      ; BEGIN is the starting execution address
```

ENDBUF End Buffer

Marks the end of a buffer block.

```
ENDBUF
```

Remarks

When the assembler encounters this directive, the location counter points just beyond the end of the buffer. You may not use a label with this directive.

Related Directive

BUFFER

Example

```
ORG P:$100
BUFFER R,64      ; Uninitialized reverse-carry buffer
ENDBUF
```

ENDIF End Conditional Assembly

Ends conditional assembly that the preceding IF directive began.

```
ENDIF
```

Remarks

The directive pair IF and ENDIF delimit source lines for conditional assembly. You can nest conditional assembly directives to any level. The ENDIF directive, like the optional ELSE directive, always pairs with the closest previous IF directive.

You may not use a label with this directive.

Related Directives

ELSE, IF

Example

```
IF @REL()
SAVEPCSET *      ; Save current program counter
ENDIF
```

ENDM End Macro Definition

Ends a macro definition or marks the end of duplicated lines: terminates assembly actions of the MACRO, DUP, DUPA, DUPC, or DUPF directives. You may not use a label with this directive.

```
ENDM
```

Related Directives

DUP, DUPA, DUPC, DUPF, MACRO

Example

```
SWAP_SYM  MACRO REG1,REG    ;Swap REG1,REG2; D4.L is
temp
          MOVE  R\?REG1,D4.L
          MOVE  R\?REG2,R\?REG1
          MOVE  D4.L,R\?REG2
           ENDM
```

Freescale Semiconductor, Inc.
SC_Assem.book Page 121 Thursday, September 23, 2004 10:44 AM

ENDSEC End Section

Marks the end of a section; the previous SECTION directive began the section. You may not use a label with this directive.

```
ENDSEC
```

Related Directive

SECTION

Example

```
          SECTION .data
VALUES    BSC $100        ; Initialize to zero
           ENDSEC
```

EQU Equate Symbol to Value

Assigns an expression value to a label symbol.

```
label   EQU [L:|N:|P:]expression
```

Parameters

`label`

Label that receives the `expression` value.

`L:|N:|P:`

Optional specifier that forces the memory-space attribute; valid only if the expression memory-space attribute is N (none).

`expression`

Any absolute or relative expression; must not include any forward references.

Remarks

Many directives assign the program-counter value to a label; this directive gives an expression value to the specified label. You cannot redefine this label anywhere in the program or section.

The optional forcing memory space parameter lets you assign an attribute to a constants-only expression that refers to a fixed address in a memory space. However, if the `expression` attribute is L or P and you specify an attribute that does not match, the assembler issues an error message.

For More Information: www.freescale.com

Examples

This first example assigns the value $4000, and memory-space attribute P, to the symbol A_D_PORT:

```
A_D_PORT  EQU P:$4000
```

This second example gives symbol COMPUTE the value and memory-space attribute of the expression @LCV(L):

```
COMPUTE   EQU @LCV(L)
```

EXITM Exit Macro

Immediately terminates a macro expansion or a sequence of duplicated lines.

```
EXITM
```

Remarks

Use this directive with conditional-assembly directives to terminate macro expansion (or duplicated lines) upon detection of an error condition. You may not use a label with this directive.

Related Directives

DUP, DUPA, DUPC, DUPF, MACRO

Example

```
CALC    MACRO  XVAL,YVAL
        IF  XVAL<0
        FAIL  'Macro parameter value out of range'
        EXITM                                  ; Exit
macro
        ENDIF
        .
        .
        .
         ENDM
```

FAIL Issue Programmer Error Message

Issues the specified error messages and increments the total error count.

```
FAIL  {str|exp}[,{str|exp},...]
```

Freescale Semiconductor, Inc.
SC_Assem.book Page 123 Thursday, September 23, 2004 10:44 AM

Parameters

`str`

Any valid string appropriate as part of an error message. Commas without spaces must separate multiple `str` values.

`exp`

Any expression appropriate as part of an error message. Commas without spaces must separate multiple `exp` values.

Remarks

Use this directive with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the assembler displays the error message.

You may not use a label with this directive.

Related Directives

MSG, WARN

Example

```
FAIL  'Parameter out of range'
```

FALIGN Align with Fetch-Set

Aligns the address of a hardware loop's first instruction, or the address of a jump-instruction target, with the fetch set. The fetch set boundary is 16 bytes for the SC140 core.

`FALIGN`

Remarks

You may apply the FALIGN directive on a per-loop basis. The assembler performs alignment if the size of the execution set overlaps the fetch set boundary. If the execution set starts at a nonaligned address, but fits into the current fetch set, the assembler does not perform alignment.

The assembler implements alignment by padding:

- If you optimize the function for speed (OPT_SPEED — the default), the assembler inserts the appropriate number of NOPs inside packets, before the loop that contains the FALIGN directive.

For More Information: www.freescale.com

- If you optimize the function for size (OPT_SIZE), the assembler inserts the appropriate number of NOPs as a standalone packet.

Any time the assembler inserts NOPS, it issues a remark.

For the SC110 core, the FALIGN directive forces its entire section to an 8-bit alignment. For the SC120, SC140, SC1200, or SC1400 cores, the FALIGN directive forces its entire section to a 16-byte alignment. The system preserves these alignments at link time, even if the section starting location has moved.

Related Directive

OPT (LPA option)

Example

```
        ORG P:$100
        DOSETUP3 compute_alpha
        DOEN3  #5
        NOP
        NOP
        NOP
        LOOPSTART3
         FALIGN
compute_alpha
          ...
           LOOPEND3
```

GLOBAL Declare Global Section Symbol

Makes specified local section symbols global. (The default arrangements for such symbols is that they are local.) You may not use a label with this directive.

```
GLOBAL  symbol[,symbol,...]
```

Parameter

`symbol`

Any symbol defined within the section, that is, between the SECTION and ENDSEC directives. Commas without spaces must separate multiple `symbol` values.

Related Directive

SECTION

Example

```
SECTION IO
GLOBAL LOOPA      ; LOOPA now globally accessible
.                 ; by other files
.
.
ENDSEC
```

GSET Set Global Symbol to Value

Assigns the specified value to the specified global symbol.

```
label    GSET value
         GSET label value
```

Parameters

`label`

A global symbol.

`value`

An absolute expression. Must not contain any forward references.

Remarks

If you use this directive to define a label, another GSET or SET directive elsewhere in the program can redefine the label. Use this directive to reset a global SET symbol within a section, where the SET symbol otherwise would be local.

Related Directives

DEFINE, EQU, SET

Example

```
COUNT   GSET 0        ; Initialize count
```

HIMEM Set High Memory Bounds

Establishes an absolute high memory bound for code and data generation. This directive is valid only for absolute mode; you may not use a label with this directive.

```
HIMEM  P:expr[,...]
```

Parameters

`P:`

P memory specifier.

`expr`

An absolute integer value within the processor address range.

Related Directive

LOMEM

Example

```
HIMEM  P:$7FFF        ; Set P run high mem limit to $7fff
```

IF Start Conditional Assembly

Starts conditional assembly of source lines: the assembler assembles the following lines if the `condition` value is true (non-zero). The ENDIF directive indicates the last source line to be assembled conditionally.

```
IF condition
.
.
[ELSE]
.
.
ENDIF
```

Parameter

`condition`

An absolute-integer expression. A non-zero value means that the condition is true; a zero value means that the condition is false. Must not contain any forward references.

Remarks

The directive pair IF and ENDIF delimit source lines for conditional assembly. You can nest conditional assembly directives to any level. The ENDIF directive, like the optional ELSE directive, always pairs with the closest previous IF directive.

The assembler follows these rules:

Freescale Semiconductor, Inc.
SC_Assem.book Page 127 Thursday, September 23, 2004 10:44 AM

- `condition` = TRUE (non-zero), no ELSE directive — assembles lines between IF and ENDIF.
- `condition` = FALSE (zero), no ELSE directive — ignores lines between IF and ENDIF.
- `condition` = TRUE (non-zero), ELSE directive present — assembles lines between IF and ELSE, ignores lines between ELSE and ENDIF.
- `condition` = FALSE (zero), ELSE directive preset — ignores lines between IF and ELSE, assembles lines between ELSE and ENDIF.

You may not use a label with this directive.

Related Directives

ELSE, ENDIF

Example

If the mode is big-endian and the assembler encounters this code:

```
IF @DEF('BIG_ENDIAN')
  MOVE.W  (r0)-,d1        ; Start traceback from
ELSE                      ;   state zero.
  MOVE.W  (r0),d1
ENDIF
```

The assembler assembles the line `MOVE.W  (r0)-,d1` and ignores the line `MOVE.W  (r0),d1`.

INCLUDE Include Secondary File

Tells the assembler to read source statements from the secondary file that the `string` argument identifies.

```
INCLUDE  {'file' | <file>}
```

Parameters

`'file' or <file>`

A file name compatible with the operating system; may include a pathname. The default extension is .asm.

Remarks

The assembler's search path depends on the file name syntax. For the `'file'` syntax, the assembler first searches in the current file (or in the specified directory,

For More Information: www.freescale.com

if the file value includes a pathname). If it does not fine the file, it next searches in all directories that the `-i` command-line option specifies.

For the `<file>` syntax, the assembler ignores the current (or specified) directory, searching only in the directories that the `-i` command-line option specifies.

You may not use a label with this directive.

Related Directive

MACLIB

Example

```
INCLUDE  'headers/io.asm'     ; Include file io.asm,
                                 ; in directory
headers.
INCLUDE  <data>               ; Include file data.asm,
                              ; but do not search in the
                                 ; current directory
```

LIST List Assembly

Prints the source listing: all lines that follow the LIST directive.

```
LIST
```

Remarks

The printed source listing begins with the line after the LIST directive.

A special list counter, initialized to 1, affects list printing: as long as the counter value remains above zero, the assembler prints the source listing in response to any LIST directive. Each LIST directive increments the list counter, but each NOLIST directive decrements the counter. If the counter value drops to zero or below, the assembler does not print a listing in response to the LIST directive.

If the command line includes the `-oIL` option, the assembler ignores the LIST directive, regardless of the counter value.

You may not use a label with this directive.

Related Directives

NOLIST, OPT

Example

```
IF  LISTON
```

```
        LIST                ; Turn the listing back on
        ENDIF
```

LOMEM Set Low Memory Bounds

Establishes an absolute low memory bound for code and data generation. This directive is valid only for absolute mode; you may not use a label with this directive.

```
LOMEM  P:expr[,...]
```

Parameters

`P:`

P memory specifier.

`expr`

An absolute integer value within the processor address range.

Related Directive

HIMEM

Example

```
HIMEM  P:$100        ; Set P run low mem limit to $100
```

MACLIB Specify Macro Library

Specifies a directory that contains macro definitions.

```
MACLIB  pathname
```

Parameter

`pathname`

Pathname of a macro-definition directory.

Remarks

Each macro definition must be in a separate file; the file must have the same name as the macro, with extension `.asm`. So file `blockmv.asm` would contain the definition of macro `blockmv`.

Upon encountering a directive that is not in the directive or mnemonic tables, the assembler searches the directory that `pathname` specifies. If it finds a matching

file, the assembler saves the current source line, then opens the file for input as an INCLUDE file. When the assembler reaches the end of the file, it restores the source line and resumes processing.

If the processed file does not include a macro definition of the unknown directive, the assembler issues an error message upon restoring and processing the source line. However, the processed file is not limited to macro definitions; it can include any valid source code statements.

If multiple MACLIB directives specify several directories, the assembler searches the directories in their order in the MACLIB directives.

You may not use a label with this directive.

Related Directive

INCLUDE

Example

```
MACLIB  'macros/mymacs/'
```

MACRO Define Macro

Defines a new macro.

```
label    MACRO [dumarg[,dumarg,...]]
              .
              .
             source statements
              .
              .
               ENDM
```

Parameters

`label`

Name for the new macro; should not duplicate any existing assembler directives or mnemonics.

`dumarg`

Symbolic name to be replaced by an argument value when a statement calls the macro. Must follow the rules for symbol names; may not begin with the % character. Commas without spaces must separate multiple dumarg values.

Remarks

Each macro definition consists of three parts:

- Header — the name and dummy arguments. This is the MACRO directive, with its `label` and `dumarg` values. (The assembler issues a warning if the label value duplicates an assembler directive or mnemonic.)
- Body — the sequence of standard source statements.
- Terminator — the ENDM directive.

If you nest macro definitions, the assembler does not define the nested macro until it expands the primary macro.

Related Directives

DUP, DUPA, DUPC, DUPF, ENDM

Example

```
SWAP_SYM  MACRO REG1,REG2                ; Swap
REG1,REG2;
          MOVE R\?REG1,D0                ; use D0 as temp
          MOVE R\?REG2,R\?REG1
          MOVE D0,R\?REG2
           ENDM
```

MSG Issue Programmer Message

Outputs a message without incrementing the error or warning counts.

```
MSG  {str|exp}[,{str|exp},...]
```

Parameters

`str`

Any valid string appropriate as part of a message. Commas without spaces must separate multiple `str` values.

`exp`

Any expression appropriate as part of a message. Commas without spaces must separate multiple `exp` values.

Remarks

Use this directive with conditional assembly directives to convey information. The assembly proceeds normally after the assembler displays the message.

You may not use a label with this directive.

Related Directives

FAIL, WARN

Example

```
MSG  'Generating sine tables'
```

NOLIST Stop Assembly Listing

Stops printing of the assembly listing.

```
NOLIST
```

Remarks

A special list counter, initialized to 1, affects list printing: as long as the counter value remains above zero, the assembler prints the source listing in response to any LIST directive. Each LIST directive increments the list counter, but each NOLIST directive decrements the counter. If the counter value drops to zero or below, the assembler does not print a listing in response to the LIST directive.

You may not use a label with this directive.

Related Directives

LIST, OPT

Example

```
IF  LISTOFF
NOLIST          ; Turn the listing off
ENDIF
```

NOTE Include Note

Tells the assembler to include the specified note in the .note section of the listing file.

```
NOTE"<comment>"
```

Freescale Semiconductor, Inc.
SC_Assem.book Page 133 Thursday, September 23, 2004 10:44 AM

Parameter

`<comment>`

User-specified note or comment.

Example

```
NOTE"This is a note"
```

OPT Set Assembler Options

Specifies options that control formats, reporting, and other aspects of assembler operation.

```
OPT option[,option...]      [; comment]
```

Parameters

`option`

Any assembler control option: valid either with the OPT directive or with the -o command line option. Table X lists these options.

`comment`

Optional comment string.

Remarks

Commas without spaces must separate multiple options. You may not use a label with this directive.

The same options can be arguments for this directive or for the `-o` command line option: `OPT MU` in a source line has the same effect as -o in the command line.

Table 5.3 lists available options, by type. The table shows the options in capital letters, but the options are not case sensitive. Many options have logical opposites that begin with NO; table explanations include these opposites.

For More Information: www.freescale.com

Table 5.3 OPT Options

Type	Option	Control Action
Listing Format Control	FC	Folds trailing comments under the source line, aligning them with the opcode field; aligns lines that start with the ; character with the label field. NOFC option: Does not fold trailing comments. Default setting; reset setting at end of pass one.
	FF	Uses form feeds for page ejects in the source listing. NOFF option: Uses multiple line feeds for page ejects. Default setting; reset setting at end of pass one.
	FM	Formats messages: aligns text, breaks at word boundaries. NOFM option: Does not format messages. Default setting; reset setting at end of pass one.
	PP	Pretty prints: aligns values in consistent columns, without regard to source-file formatting. Default setting; reset setting at end of pass one. NOPP option: Does not pretty print; preserves source-line formatting, but expands tabs to spaces and concatenates continuation lines.
	RC	Uses relative comment spacing: lets comment starting column float, according to presence/absence of other field values. NORC option: Does not use relative spacing. Default setting; reset setting at end of pass one.
Output file format	COFF	Specifies a COFF binary output file.
	ELF	Specifies an ELF binary output file.
	REL	Specifies a .rel relocation type.
	RELA	Specifies a .rela relocation type.
Reporting	CEX	Prints DC expansions. NOCEX option: Does not print DC expansions. Default setting; reset setting at end of pass one.

Table 5.3 OPT Options (*continued*)

Type	Option	Control Action
	CL	Prints conditional assembly directives. Default setting; reset setting at end of pass one. NOCL option: Does not print conditional assembly directives.
	DXL	Expands DEFINE directive strings in the source listing. Default setting; reset setting at end of pass one. NODXL option: Does not expand DEFINE directive strings.
	HDR	Generates listing header along with titles and subtitles. Default setting; reset setting at end of pass one. NOHDR option: Does not generate listing header; turns off titles and subtitles.
	IL	Inhibits (blocks) a source listing.
	MC	Prints macro calls. Default setting; reset setting at end of pass one. NOMC option: Does not print macro calls.
	MD	Prints macro definitions. Default setting; reset setting at end of pass one. NOMD option: Does not print macro definitions.
	MEX	Prints macro expansions. NOMEX option: Does not print macro expansions. Default setting; reset setting at end of pass one.
	MU	Includes a memory utilization report in the source listing. This option must appear before any code or data generation.
	NL	Prints nesting levels in listing: conditional assembly and section nesting. NONL option: Does not print nesting levels. Default setting; reset setting at end of pass one.
	WEX	Counts warnings as error messages. Warnings block creation of an object file, unless `-osvo` is in the command line.

Table 5.3 OPT Options (*continued*)

Type	Option	Control Action
	U	Prints unassembled lines of conditional-assembly code. NOU option: Does not print unassembled, conditional-assembly lines. Default setting; reset setting at end of pass one.
Message Control	AE	Checks address expressions: validates arithmetic operations. Default setting; reset setting at end of pass one. NOAE option: Do not check address expressions.
	MSW	Issues a memory space warning if incompatibilities exist. Default setting; reset setting at end of pass one. NOMSW option: Suppresses memory space incompatibility warnings.
	UR	Warns about each unresolved reference at assembly time; valid only in relocatable mode. NOUR option: Does not warn about unresolved references.
	W	Prints all warnings. Default setting; reset setting at end of pass one. NOW option: Does not print warnings.
Symbol	DEX	Expands DEFINE symbols in quoted strings. To expand individual symbols, use double-quoted strings. NODEX option: Does not expand DEFINE symbols in quoted strings. Default setting; reset setting at end of pass one.
	IC	Ignores case for symbols, section names, and macro names; must appear before any definitions of symbols, sections, and macros.
	SO	Writes symbol information to the object file.
Assembler Operation	AEC	Permits standard C-language escape characters in DC directives; changes DC directive default setting. NOAEC option: do not allow C escape characters. Default setting.
	BE	Generates output for a big-endian target. (The default setting is little-endian.)

Table 5.3 OPT Options (*continued*)

Type	Option	Control Action
	CC	Enables cycle counts, clears the total cycle count. The output listing shows cycle counts for each instruction, as if there were a full instruction fetch pipeline and no wait states. NOCC option: Disables cycle counts, does not clear the total cycle count. Default setting; reset setting at end of pass one.
	CK	Enables checksumming for instructions and data values; clears cumulative checksum. NOCK option: Disables checksumming. Default setting; reset setting at end of pass one.
	CM	Preserves comment lines of macro definitions; does not preserve comment lines that begin with ;; characters. NOCM option: Does not preserve comment lines of macro definitions.
	CONTCK	Re-enables checksumming for instructions and data values; does not clear cumulative checksum.
	CS	Classifies carry-update instructions as irrelevant, letting the assembler re-order instructions of a packet for minimum length.
	DLD	Does not restrict DO-loop directives; suppresses error messages regarding directives in DO loops that do not make sense. NODLD option: Restricts DO-loop directives. Default setting; reset setting at end of pass one.
	GL	Makes all symbols global; must appear in the source file before any explicit section definitions.
	INTR	Checks interrupt locations; lets the assembler screen interrupt vector locations of memory for inappropriate DSP instructions. Default setting; reset setting for absolute mode. NOINTER option: Does not check interrupt locations. Reset setting for relative mode.
	LDB	Enables source listing debug: using the source listing instead of the assembly language file as the debug source file. Valid only if the command line includes the `-l` option.
	LE	Enables little-endian mode. Default setting.

Table 5.3 OPT Options (*continued*)

Type	Option	Control Action
	LPA	Enables hardware loop alignment for subsequent loops of the file; aligns loop addresses on a fetch set boundary, preventing additional pipeline stall cycles. The assembler implements this alignment automatically by padding execution sets with NOPs. However, this padding does not take place in hardware loops that do not use LOOPSTART notation. NOLPA option: Disables hardware loop alignment. Default setting.
	MB	Specifies the big memory model: move instruction addresses use the {a32} format instead of the default {a16} format.
	MI	Expands scanning for macro include files to all MACLIB directories; overrides directory restrictions of INCLUDE directive, MACLIB directive, and `-i` command-line option. NOMI option: Does not expand scanning. Default setting; reset setting at end of pass one.
	NONMCHK	Blocks checking for reserved symbol name violations.
	NOOVLDBG	For files containing an overlay section, renames debug sections .debug_info<overlay_scn>. (In this format, multiple overlay sections in the same module corrupt the debug information.)
	OVLDBG	Default setting. Assembler generates overlay-section debug information using local addresses instead of run addresses. This mode permits debugging of modules that contain multiple overlay sections.
	PS	Packs strings for the DC directive; packs individual string bytes into consecutive target words. NOPS option: Does not pack strings; stores one string byte per word.
	SC110	Sets target architecture for the SC110 DSP core (one MAC unit).
	SC120	Sets target architecture for the SC120 DSP core (two MAC units).
	SC1200	Sets target architecture for the SC1200 DSP core (two MAC units)/
	SC140	Sets target architecture for the SC140 DSP core (four MAC units. Default setting.
	SC140E	Sets target architecture for the SC140E DSP core (four MAC units.)

Table 5.3 OPT Options (*continued*)

Type	Option	Control Action
	SC1400	Sets target architecture for the SC1400 DSP core (four MAC units).
	SRB	Specifies registers B0—B7 instead of R8—R15 for PUS, POP, and MOVE instructions.
	SVO	Preserves object file if errors: overrides default setting to delete object files in case of errors. Must appear before any code or data generation.

Example

```
OPT CEX,MEX    ; Enable DC and macro expansions
OPT CRE,MU       ; Print cross reference, memory
utilization
```

ORG Initialize Memory Space and Location Counters

Sets absolute addresses; sets the memory space; and specifies and initializes the location counter.

```
ORG P[{lc|(ce)}]:[initval]
```

Parameters

`P`

P memory specifier.

`lc`

Location counter specifier: L (counter 1) or H (counter 2). Omitting both values specifies the default counter (counter 0) .

Counters are useful for providing mnemonic links among individual memory blocks.

`(ce)`

The counter number: a non-negative absolute integer expression. Must be in parentheses, must not exceed 65535.

`initval`

Optional initial value for the location counter. If you omit this value, the assembler uses the last value of the counter.

Remarks

Counters provide mnemonic links among individual memory blocks. The H, L, and default counter names are symbolic; the assembler does not verify that the H-counter value is greater than the L-counter value.

You may not use a label with this directive.

Examples

This first example:

```
ORG  P:$1000
```

Sets the memory space to P, selects the P-space default counter, and initializes that counter to $1000.

This second example:

```
ORG  PH:
```

Sets the memory space to P and selects the P-space H location counter (counter 2). This example does not initialize the counter, so the assembler uses the last H-counter value.

PAGE Advance Page or Size Page

Without argument values, advances the source listing to the top of the next page. With argument values, sets the size and margins of source-listing pages.

PAGE [*pagewidth*[,*pagelength*,*blanktop*,*blankbtm*,*blankleft*]]

Parameters

`pagewidth`

Number of columns per line: 1 through 255. The default is 80.

`pagelength`

Number of lines per page: 10 through 255. The default is 66. The special value 0 turns off all headers, titles, subtitles, and page breaks.

`blanktop`

Number of blank lines at the top of the page. The minimum (and default) value is 0. The value must maintain the relationship:
`blanktop+blankbtm f pagelength-10`.

Freescale Semiconductor, Inc.
SC_Assem.book Page 141 Thursday, September 23, 2004 10:44 AM

`blankbtm`

Number of blank lines at the bottom of the page. The minimum (and default) value is 0. The value must maintain the relationship:
`blanktop+blankbtm ≤ pagelength-10.`

`blankleft`

Number of blank columns at the left of the page; must be less than the `pagewidth` value. The minimum (and default) value is 0.

Remarks

If this directive has no arguments, the assembler does not print the directive in the source listing. If this directive does have argument values, the assembler includes the directive in the source listing.

Arguments may be any non-negative absolute integer expressions; commas without spaces must separate multiple argument values. Two adjacent commas tell the assembler to use the default value (or the last set value).

You may not use a label with this directive.

Examples

This first example:

```
PAGE  132,,2,4
```

Sets the page width to 132 columns, page length to 66 lines (the default), top margin to 2 lines, and bottom margin to 4 lines. This example specifies 0 blank columns (the default) at the left side of the page.

This second example:

```
PAGE
```

merely advances the listing to the top of the next page.

PMACRO Purge Macro Definition

Purges the specified macro definitions from the macro table, reclaiming space from the table. You may not use a label with this directive.

```
PMACRO  symbol[,symbol,...]
```

Parameter

`symbol`

A macro name. Commas without spaces must separate multiple `symbol` values.

For More Information: www.freescale.com

Related Directive

MACRO

Example

```
PMACRO  MAC1,MAC2
```

PRCTL Send Control String to Printer

Concatenates its arguments and sends them to the source listing, provided that the command line included the `-l` option.

```
PRCTL  {exp | string},...
```

Parameters

`exp`

A byte expression that encodes non-printing control characters, such as ESC.

`string`

Any valid assembler control string.

Remarks

You may use this directive anywhere in the source file; the assembler outputs the control string at the corresponding place in the source listing. If a PRCTL directive is the first line of the first input file, the assembler outputs the control string before outputting page headings or titles.

If a PRCTL directive is the last line of the last input file, the assembler makes sure that all error summaries, symbol tables, and cross-references have been printed before it prints the control string. In this manner, you can use a PRCTL directive to restore the previous printer mode once printing is done.

If the command line did not include the `-l` option, the assembler ignores this directive. The assembler does not print this directive unless there is an error.

The assembler does not allow a label with this directive.

Example

```
PRCTL $1B,'E'          ; Reset HP LaserJet printer
```

RADIX Set Constant Radix

Sets the specified number base (radix) for constants.

```
RADIX  value
```

Parameter

`value`

An absolute integer expression for the value 2, 10, or 16. The default value is 10.

Remarks

This directive lets you specify the radix for constants that do not have a leading radix indicator. The leading radix indicator for decimal numbers is the grave accent (‘). If you use a constant to change the radix, it must have the appropriate number base when the assembler encounters the RADIX directive.

You may not use a label with this directive.

Example

```
_RAD10   DC 10        ; Evaluates to hex A
         RADIX 2
_RAD2    DC 10        ; Evaluates to hex 2
         RADIX `16
_RAD16   DC 10        ; Evaluates to hex 10
          RADIX 3       ; Bad radix expression
```

SECFLAGS Set ELF Section Flags

Sets flag bits for the current section.

```
SECFLAGS  flag[,flag...]
```

Parameter

`flag`

Any of these attribute values:

`write` — section contains writable data when loaded

`alloc` — section occupies memory space when loaded

`execinstr` — section contains executable instructions

For More Information: www.freescale.com

`nowrite` — section contains non-writable data when loaded

`noalloc` — section does not occupy memory space when loaded

`noexecinstr` — section does not contain executable instructions

Remarks

Commas without spaces must separate multiple `flag` arguments.

Conventional sections (.text, .data, .rodata, or .bss) have specific type and flag values. A section with any other name is a code section; its default type and flag values are those of a .text section. But you can use the SECFLAGS and SECTYPE directives to redefine the default values.

You may not use a label with this directive.

Related Directives

SECTYPE, SECTION

Example

The SECTION directive begins a data section that has a non-standard name. Accordingly, the assembler gives this section the default flag values of a .text section: nowrite, alloc, execinstr. The SECFLAGS directive makes the flag values appropriate for a data section.

```
SECTION .data_input2
SECFLAGS write,alloc,noexecinstr
...
ENDSEC
```

SECTION Start Section

Starts a section: a block of relocatable code or data.

```
SECTION  symbol [GLOBAL] [core_id]
      .
      .
      section source statements
      .
      .
      ENDSEC
```

Freescale Semiconductor, Inc.
SC_Assem.book Page 145 Thursday, September 23, 2004 10:44 AM

Parameters

`symbol`

Name for the section. Standard names are `.text`, `.data`, `.rodata`, and `.bss`. Other names automatically invoke .text type and attribute values; other names must not duplicate a reserved name.

`GLOBAL`

Optional qualifier that makes all symbols defined within the section global. Without this parameter, all symbols defined within the section are local.

`core_id`

Optional 8102 DSP core loading destination for the section. Does not pertain to other processors.

Remarks

Code or data inside a section is independently relocatable within the memory space to which it is bound.

You can nest sections to any level. When the assembler encounters a nested section, it stacks the parent section and uses the nested section. Upon encountering the nested section's ENDSEC directive, the assembler restores and uses the parent section. (The ENDSEC directive always pairs with the closest previous SECTION directive.)

You can split a section into separate parts, by using the same name for multiple SECTION and ENDSEC pairs. This lets you arrange program source statements arbitrarily. For example, you can group all statements that reserve P space storage locations.

The assembler allocates a P-memory-space location counter for every section you define in source code. This counter maintains data and instruction offsets from the beginning of the section. At link time, according to your specifications, the linker relocates sections to an absolute address, loads sections in a particular order, or linked sections contiguously. The linker logically recombines split sections, making it possible to relocate each section as a unit.

You may give a section any name, except for the reserved names of Table 5.4.

Table 5.4 Reserved Section Names

.debug_abbrev	.debug_pubname	.rel.line
.debug_aranges	.default	.rel.line.debug_info
.sc100.delay_slots	.hash	.rel.text
.debug_info	.line	.shstrtab

For More Information: www.freescale.com

Table 5.4 Reserved Section Names

.debug_line	.mw_info	.strtab
.debug_loc	.note	.symtab
.debug_macinfo	.rel.sc100.delay_slots	
.debug_macro	.rel.debug_loc	

Table 5.5 explains the standard sections.

Table 5.5 Conventional ELF Sections

Section	Contents	Type	Attributes
.bss	Unititialized data	NOBITS	ALLOC, WRITE
.data	Initialized data	PROGBITS	ALLOC, WRITE
.mw_info	Assembler-generated contents that the linker consumes during dead data stripping	SHT_MW_INFO (SHT_LOPROC+3)	no sh_flags(0)
.note	User comments, as ABI 2.0 defines.	SHT_NOTE(7)	no sh_flags(0)
.rodata	Read-only initialized data	PROGBITS	ALLOC
.text	Program code	PROGBITS	ALLOC, EXECINSTR

If you do not use a standard name, the assembler assigns the .text type and attributes. To change these values, use the SECTYPE or SECFLAGS directives.

Symbols defined outside any section are global: they can satisfy an outstanding reference in the current file at assembly time, or in any file at link time. You may reference global symbols freely from inside or outside any section, as long as the global symbol name is unique.

Symbols defined within a section are local: they can satisfy an outstanding reference only within that section. But you can change this default arrangement:

- The GLOBAL qualifier of the SECTION directive makes all symbols defined in that section global symbols.
- The GLOBAL directive makes specified symbols global.

For the 8102 DSP, the linker can generate four linked core files, one for each processor, one of which contains the L2 memory. You must specify the core file

Freescale Semiconductor, Inc.
SC_Assem.book Page 147 Thursday, September 23, 2004 10:44 AM

into which the system will load the section. One way is to include a `core_id` specifier in the SECTION directive. The other method is to use the linker command file.

You man not use a label with this directive.

Related Directives

ORG, GLOBAL

Example

This directive starts a new section, TABLES:

```
SECTION TABLES GLOBAL
```

As the section has a non-standard name, its type and attributes are those of a .text section. The GLOBAL specifier means that all symbol definitions in the section define global symbols.

SECTYPE Set ELF Section Type

Defines the section type.

```
SECTYPE {progbits|nobits|overlay}
```

Parameters

`progbits`

Specifier for a section that has program contents, including code and data.

`nobits`

Specifier for a section that has no contents and does not occupy file space. (The assembler discards anything in `nobits` sections.)

`overlay`

Specifier for an overlay section.

Remarks

Standard sections `.text`, `.data`, `.rodata`, and `.bss` have default type and flag values; a section that has any other name receives the `.text` default values. Use this directive to change the default type value.

You may not use a label with this directive.

For More Information: www.freescale.com

Related Directives

SECFLAGS, SECTION

Example

The SECTION directive starts new section `.data_output`. The non-standard name means that the section receives `.text` default type and attributes: PROGBITS, ALLOC, and EXECINSTR. The SECFLAGS directive changes the attributes; the SECTYPE directive changes the type to NOBITS.

```
SECTION .data_output
SECFLAGS write,alloc,noexecinstr
SECTYPE nobits
...
ENDSEC
```

SET Set Symbol to Value

Assigns the specified value to the specified symbol.

```
label    SET value
         SET label value
```

Parameters

`label`

A symbol.

`value`

An absolute expression. Must not contain any forward references.

Remarks

If you use this directive to define a label, another SET directive elsewhere in the program can redefine the label. Use this directive to establish temporary or reusable counters within a macro.

Related Directives

DEFINE, EQU, GSET

Example

```
COUNT   SET 0        ; Initialize count
```

Freescale Semiconductor, Inc.
SC_Assem.book Page 149 Thursday, September 23, 2004 10:44 AM

SIZE Set Symbol Size

Sets the size of the specified symbol to the value of the expression parameter. May be anywhere in the source file, unless the symbol is a function name. If you define an INITIALIZER or VARIABLE symbol, the SIZE directive should appear after the symbol definition.

```
SIZE  symbol,expression[,alignment]
```

Parameters

`symbol`

Any valid symbol. If a function name, the function definition must precede the SIZE directive.

`expression`

Any valid expression.

`alignment`

Optional alignment value for the symbol in the `.mw_info` section. (The linker uses this section's information for dead stripping.)

Related Directive

TYPE

Example

```
_main:
.
.
RTS
SIZE  _main,(*-_main)
```

STITLE Initialize Program Subtitle

Makes the specified string a subtitle of the program.

```
STITLE  [string]
```

Parameter

`string`

Optional string value.

For More Information: www.freescale.com

Remarks

The default program subtitle at the top of source-listing pages is blank. This directive specifies the subtitle for subsequent pages of the source listing. A subsequent STITLE directive changes the subtitle again. An STITLE directive without any string argument makes the subtitle blank.

The source listing does not include this directive.

You may not use a label with this directive.

Related Directive

TITLE

Example

```
STITLE  'COLLECT SAMPLES'
```

TITLE Initialize Program Title

Makes the specified string the title of the program.

```
TITLE  [string]
```

Parameter

`string`

Optional string value.

Remarks

The default program title at the top of source-listing pages is blank. This directive specifies the title for subsequent pages of the source listing. A subsequent TITLE directive changes the title again. A TITLE directive without any string argument makes the title blank.

The source listing does not include this directive.

You may not use a label with this directive.

Related Directive

STITLE

Example

```
TITLE  'FIR FILTER'
```

TYPE Set Symbol Type

Sets the type for the specified symbol.

```
Label   TYPE  typeid
```

Parameters

`Label`

Any label symbol of the program

`typeid`

Any of these values:

`FILE` — for the file name of the compilation unit.

`FUNC` — for a symbol associated with a function or other executable code.

`INITIALIZER` — for a symbol associated with an initializer.

`OBJECT` — for a symbol is associated with a variable, array, structure, or other such object.

`VARIABLE` — for a symbol associated with a variable.

Remarks

The assembler stores INITIALIZER- and VARIABLE- type information in the .mw_info section. The linker uses this information for data dead stripping. The linker stripping support document gives additional information about these initializers and variables.

Related Directive

SIZE

Example

```
Afunc   TYPE  FUNC        ; Symbol Afunc is type
STT_FUNC
```

UNDEF Undefine DEFINE Symbol

Cancels the substitution string for the specified DEFINE symbol.

```
UNDEF  symbol
```

Parameter

`symbol`

Any symbol that a previous DEFINE directive specified.

Remarks

A previous DEFINE directive specified a substitution string for the symbol. This directive releases that substitution string; `symbol` no longer represents a valid DEFINE substitution.

You may not use a label with this directive.

Related Directive

DEFINE

Example

```
UNDEF   DEBUG    ; Undefine the debug substitution
string
```

WARN Issue Programmer Warning

Outputs a warning, incrementing the warning count.

`WARN  {str|exp}[,{str|exp},...]`

Parameters

`str`

Any valid string appropriate as part of a warning. Commas without spaces must separate multiple `str` values.

`exp`

Any expression appropriate as part of a warning. Commas without spaces must separate multiple `exp` values.

Remarks

Use this directive with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the assembler displays the warning.

You may not use a label with this directive.

Related Directives

FAIL, MSG

Example

```
WARN  'Parameter value too large'
```

XDEF Define External Symbol

For COFF mode only, makes symbols defined in the current section accessible to sections that have the corresponding XREF directive.

```
XDEF  symbol[,symbol,...]
```

Parameter

`symbol`

Symbol defined in the current section, to be made accessible to other sections. Commas without spaces must separate multiple `symbol` values.

Remarks

This directive pertains to COFF mode only; ELF mode does not support this directive.

This directive must appear between SECTION and ENDSEC directives; it must precede the definitions of the specified symbols.

You may not use a label with this directive.

Related Directives

SECTION, XREF

Example

```
SECTION IO
XDEF  LOOPA           ; LOOPA valid in sections with
XREF
.
.
.
ENDSEC
```

XREF Reference External Symbol

For COFF mode only, makes symbols defined elsewhere valid in the current section.

```
XREF  symbol[,symbol,...]
```

Parameter

`symbol`

Symbol to be valid in the current section. The symbol definition must be outside any section, or must be declared global by an XDEF directive. Commas without spaces must separate multiple `symbol` values.

Remarks

This directive pertains to COFF mode only; ELF mode does not support this directive.

This directive must appear between SECTION and ENDSEC directives; it must precede any references to the specified symbols.

You may not use a label with this directive.

Related Directives

SECTION, XDEF

Example

```
SECTION FILTER
XREF  AA,CC,DD            ; Symbols valid in this section
.
.
.
ENDSEC
```

6

Macros and Conditional Assembly

Macros streamline repeated patterns of code or groups of instructions. If you define such a pattern as a macro, you can call the macro at appropriate program locations instead of repeating the pattern.

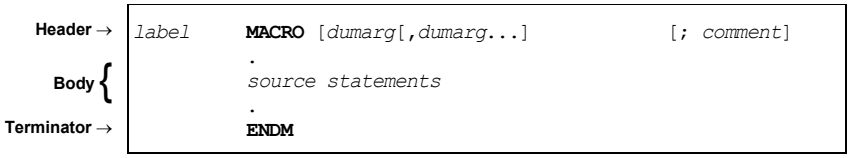
For some patterns, variable values change for each pattern repetition. Other patterns involve conditional assembly. Macros accommodate either case: they let you designate selected statement fields as variable. You can call such a macro as many times as necessary, substituting different parameters for the variable fields.

Defining a Macro

Before you can use a macro, you must define it, either in the source file or in a macro library. Figure 6.1 depicts a macro definition, which consists of these parts:

- Header — the MACRO directive, which assigns the name and defines dummy arguments.
- Body — the code and instructions the assembler uses for each macro call.
- Terminator — the ENDM directive.

Figure 6.1 Macro Definition



The header, or MACRO directive, includes these parameters:

`label`

Name for the new macro. If this name duplicates any existing assembler directives or mnemonic opcodes, the assembler substitutes the macro for the directive or mnemonic opcode and issues a warning. This replacement does not happen if the label value duplicates a library macro name.

For More Information: www.freescale.com

`dumarg`

Optional symbolic name to be replaced by an argument value when a statement calls the macro. Each `dumarg` value must follow the rules for global symbol names, and must not begin with the % character. Commas without spaces must separate multiple `dumarg` values.

`comment`

An optional comment.

Note that a macro definition can call other macros. Such other macros are *nested*. The definition of a nested macro must precede its appearance in a source-statement operation field. The assembler does not process calls or definitions of nested macros until it expands the parent macro.

Calling a Macro

Use a macro call to invoke a macro. In response, the assembler produces in-line code from the macro's statements, then inserts that code in the normal flow of the program. For every macro call, execution of the generated source statements takes place with execution of the rest of the program.

A macro call is a source statement that has this format:

```
[label]  macro  [arg[,arg...]]        [; comment]
```

where:

`label`

An optional label that corresponds to the location-counter value at the start of the macro expansion.

`macro`

The name of the macro; must be in the operation field.

`arg`

An optional, substitutable argument. Commas without spaces must separate multiple arguments.

`comment`

An optional comment.

These rules apply to macro arguments:

- Arguments must correspond one-to-one with the dummy arguments of the macro definition. If the call does not have the same number of arguments as the definition, the assembler issues a warning.

Freescale Semiconductor, Inc.
SC_Assem.book Page 157 Thursday, September 23, 2004 10:44 AM

- Arguments can be quoted strings, although the assembler does not require single quotes around macro argument strings. However, single quotes must surround any embedded comma or space in an argument string.
- To declare a null argument for a macro call enter two commas without any intervening spaces; declare a null string for the argument; or terminate the argument list with a comma, omitting the rest of the argument list.
- The assembler does not substitute any characters in generated statements that reference a null argument.

Macro Expansions

The assembler generates source statements in response to a macro call. These source statements are a *macro expansion*. Macro expansions may contain substitutable arguments, and their types are relatively unrestricted. They can include any processor instruction, almost any assembler directive, or any previously-defined macro. Macro-expansion source statements must conform to the same conditions and restrictions that apply to statements that a programmer writes.

Macro Libraries

Macro libraries are directories of macro definitions. Each definition must be in a separate file, each file has the macro's name with the extension `.asm`. For example, `blockmv.asm` is the file that contains the definition of macro `blockmv`.

The MACLIB directive specifies a macro library directory. Upon encountering an unknown directive, the assembler first searches for the definition in the directive and mnemonic tables. If source code includes a MACLIB directive, the assembler also searches the specified directory. If the assembler finds the `.asm` definition file in the directory, it saves the current source line and the opens the file for input as an INCLUDE file. At the end of the file, the assembler restores the source line and resumes processing.

If the processed file does not include a macro definition of the unknown directive, the assembler issues an error message upon restoring and processing the source line. However, the processed file is not limited to macro definitions: it can include any valid source code statements. If multiple MACLIB directives specify several directories, the assembler searches the directories in their order in the MACLIB directives.

Dummy Argument Operators

Table 6.1 lists text operators that permit argument text substitution during macro expansions. You can use these operators in macro definitions to concatenate text, convert numbers, and handle strings.

For More Information: www.freescale.com

Table 6.1 Macro Dummy Argument Operators

Operator	Action
\	Concatenates a macro dummy argument with adjacent alphanumeric characters.
?	For the sequence `?symbol`, substitutes a character string that represents the `symbol` decimal value.
%	Converts the sequence `%symbol` to a character string that represents the `symbol` hexadecimal value.
“	Lets you use macro arguments as literal strings.
^	Evaluates local labels at normal, not macro, scope.

Macro Directives

Directives DUP, DUPA, DUPC, and DUPF duplicate subsequent source lines. Each directive is a special macro that simultaneously defines and calls an unnamed macro. Source statements you define with any of these directives must follow the same rules as macro definitions. For the DUPA, DUPC, and DUPF directives, such source statements can include macro dummy operator characters.

Conditional Assembly

Through conditional assembly a comprehensive source program can cover many conditions. For macros, arguments specify assembly conditions. For the DEFINE, SET, and EQU directives, symbol definitions specify assembly conditions. Built-in assembler functions can test many conditions of the assembly environment.

You also can use conditional directives within a macro definition to make sure argument values are in appropriate ranges at expansion time. In this way, your macros can be self-checking and can generate error messages to any appropriate level of detail.

Use the directive pair IF and ENDIF, with the optional ELSE directive, to delimit a section of program for conditional assembly. Use this format:

```
IF condition
.
source statements
.
[ELSE]
.
```

```
source statements
.
ENDIF
```

The assembler follows these rules for conditional assembly:

- If condition = TRUE (non-zero) and there is no ELSE directive, it assembles lines between the IF and ENDIF directives.
- If condition = FALSE (zero) and there is no ELSE directive, it ignores lines between the IF and ENDIF directives.
- If condition = TRUE (non-zero) and there *is* an ELSE directive, it assembles lines between the IF and ELSE directives and ignores lines between the ELSE and ENDIF directives.
- If condition = FALSE (zero) and there *is* an ELSE directive, it ignores lines between the IF and ELSE directives and assembles lines between the ELSE and ENDIF directives.

For More Information: www.freescale.com

A

ASCII Character Codes

Table A.1 ASCII Character Codes

Decimal	Hex	ASCII	Decimal	Hex	ASCII	Decimal	Hex	ASCII
0	00	NUL	20	14	DC4	40	28	(
1	01	SOH	21	15	NAK	41	29	)
2	02	STX	22	16	SYN	42	2A	*
3	03	ETX	23	17	ETB	43	2B	+
4	04	EOT	24	18	CAN	44	2C	,
5	05	ENQ	25	19	EM	45	2D	-
6	06	ACK	26	1A	SUB	46	2E	.
7	07	BEL	27	1B	ESC	47	2F	/
8	08	BS	28	1C	FS	48	30	0
9	09	HT	29	1D	GS	49	31	1
10	0A	LF	30	1E	RS	50	32	2
11	0B	VT	31	1F	US	51	33	3
12	0C	FF	32	20	SP	52	34	4
13	0D	CR	33	21	!	53	35	5
14	0E	S0	34	22	“	54	36	6
15	0F	S1	35	23	#	55	37	7
16	10	DLW	36	24	$	56	38	8
17	11	DC1	37	25	%	57	39	9
18	12	DC2	38	26	&	58	3A	:
19	13	DC3	39	27	‘	59	3B	;

For More Information: www.freescale.com

ASCII Character Codes

Table A.1 ASCII Character Codes (*continued*)

Decimal	Hex	ASCII	Decimal	Hex	ASCII	Decimal	Hex	ASCII
60	3C	<	83	53	S	106	6A	j
61	3D	=	84	54	T	107	6B	k
62	3E	>	85	55	U	108	6C	l
63	3F	?	86	56	V	109	6D	m
64	40	@	87	57	W	110	6E	n
65	41	A	88	58	X	111	6F	o
66	42	B	89	59	Y	112	70	p
67	43	C	90	5A	Z	113	71	q
68	44	D	91	5B	[	114	72	r
69	45	E	92	5C	\	115	73	s
70	46	F	93	5D	]	116	74	t
71	47	G	94	5E	^	117	75	u
72	48	H	95	5F	_	118	76	v
73	49	I	96	60	‘	119	77	w
74	4A	J	97	61	a	120	78	x
75	4B	K	98	62	b	121	79	y
76	4C	L	99	63	c	122	7A	z
77	4D	M	100	64	d	123	7B	{
78	4E	N	101	65	e	124	7C	|
79	4F	O	102	66	f	125	7D	}
80	50	P	103	67	g	126	7E	~
81	51	Q	104	68	h	127	7F	DEL
82	52	R	105	69	i			

For More Information: www.freescale.com

B

Assembler Messages

The tables of this appendix explain assembler messages:

- Table B.1 explains command-line warnings.
- Table B.2 explains command-line error messages.
- Table B.3 explains assembler warnings.
- Table B.4 explains assembler error messages.
- Table B.5 explains assembler fatal error messages.

If possible, the assembler indicates the source field location of an error. If the assembler produces a listing file, messages usually appear immediately before the line that contains the error. One exception is an error due to the relationship between the first and last instructions of a DO loop. For such a case, the error-message text appears after the loop's last instruction. The assembler always routes messages to the standard output.

Table B.1 Command Line Warnings

Warning	Explanation
`-arch appears more than once. Using the last one`	Assembler found `-arch` option more than once on the command line. Assembler used the architecture that the last `-arch` option specified.
`Duplicate listing file specified - ignored` `Duplicate object file specified - ignored`	Assembler found `-l` or `-b` option more than once on the command line. Assembler used only the first option.
`LDB option with no listing file specified; using source file`	Command line included an `LDB` option without a corresponding `-l` (listing file) option. Without a listing file, the assembler used the default assembly language file for source tracking.
`Options for both absolute and C mode specified - C mode ignored`	Command line inappropriately had both `-a` and `-c` options. Assembler ignored the `-c` option.
`Options for both debug and strip specified - strip ignored`	Command line inappropriately had both `-g` and `-z` options. Assembler ignored the `-z` option.

For More Information: www.freescale.com

Assembler Messages

Table B.1 Command Line Warnings (*continued*)

Warning	Explanation
`Strip not valid in relocatable mode - ignored`	Command line's `-z` option was inappropriate, as the command line did not also include the `-a` option. Assembler ignored the `-z` option.
`Valid options for -arch: sc110,sc120,sc140,sc140e,msc8101 ,msc8102,pog, rainbow, starlite, sc1200, sc1400`	Invalid argument for command-line `-arch` option.
`Valid -s arguments: all,strict,none or individual restriction list`	Invalid argument for command-line `-s` option.

Table B.2 Command Line Error Messages

Message	Explanation
`-j command line option is not supported`	Assembler does not support the `-j` option for StarCore derivatives.

`Cannot open command file`
`Cannot open listing file`
`Cannot open object file`
	Assembler could not find file associated with the `-f`, `-l`, or `-b` option.

`Cannot open error file` `Cannot open output file`	Assembler could not find file associated with the `-e` or `-b` option.
`Cannot open source file`	Assembler could not find source input file.
`Cannot open the calling convention file`	Assembler could not find file that the `-k` option specified.
`Data Analysis can be performed only on 8101`	Command-line `-k` option enabled data analysis, but `-arch` option did not specify msc8101 architecture.
`Illegal architecture specified`	Command-line `-arch` option had an invalid argument.
`Illegal command line -D option argument` `Illegal command line DEFINE option`	Assembler found an error when it tried to interpret `DEFINE` parameters. Possibilities include a too-long symbol name, a symbol name that does not start with a letter, and a substitution string that lacks single quotes.

Table B.2 Command Line Error Messages (*continued*)

Message	Explanation

`Illegal command line -E option`
`Invalid syntax for command line -E option`
`Missing argument for command line -E option`
	Command-line `-e` option parameter was not `a` or `w`, included spaces, or was missing.

`Illegal command line -I option argument` `Illegal command line INCLUDE option`	Problem occurred when the assembler tried to save the include file path string.
`Illegal command line -M option argument` `Illegal command line MACLIB option`	Problem occurred when the assembler tried to save the `MACLIB` file path string.
`Illegal command line -P option argument`	Command line included obsolete `-p` option instead of current `-arch` option.
`Illegal command line -R option argument` `Illegal command line REVISION option`	Command line included obsolete `-r` option instead of current `-arch` option.
`Illegal command line option`	Command-line `-o` option was invalid.
`Interrupted`	Keyboard break (control-C) interrupted the assembler.
`Missing command line option argument`	Command line lacked appropriate arguments after an option.
`Missing option`	Command-line `-o` option lacked an argument.
`Missing source filename`	Command line lacked the source filename that should have ended the line.
`Source file name same as listing file name`	Command line specified the same filename for both source and listing files.
`Source file name same as object file name.`	Command line specified the same filename for both source and output files.

Assembler Messages

Table B.3 Assembler Warnings

Warning	Explanation
`-arch appears more than once. Using the last one`	Assembler found `-arch` option more than once on the command line. Assembler uses the architecture that the last `-arch` option specified.
`ADDA/SUBA: SP may not be a multiple of 8`	The `ADDA` or `SUBA` immediate value must be a multiple of 8; the specified SP value may not be such a multiple.
`Alignment should be a power of 2`	A symbol alignment was not a power of 2.
`BMTSET.W is not allowed on this architecture`	Specified starlite architecture is not compatible with the `BMTSET.W` instruction.
`Closing bracket without a preceding opening one`	Missing opening bracket or extra closing bracket.
`Different doen/doensh-loopstart index or missing doen/sh`	Last `doen` or `doensh` index value does not match the current loopstart index value.
`Different loopstart-loopend index`	Corresponding loopstart and loopend directives do not have the same index.
`Directive not allowed in command line absolute mode`	Command line `-a` option is active, so assembler ignored a `MOVE` directive.
`Directive not implemented in this version`	ELF-mode implementation does not include `BSM`, `BSR`, `DSM`, `DSR`, `XDEF`, or `XREF` directives.
`Duplicate listing file specified - ignored` `Duplicate object file specified -ignored`	Assembler found `-l` or `-b` option more than once on the command line. Assembler used only the first option.
`Explicit bottom margin ignored with page length of zero` `Explicit top margin ignored with page length of zero`	`PAGE` directive page-length parameter value was zero, so assembler ignored top-margin or bottom-margin parameter value.
`Expression value outside fractional domain`	Expected fractional value was outside the range -1.0 to +1.0.
`Fetch set alignment results in lpmarkb corruption. Ignoring FALIGN`	`FALIGN` attempted to insert a `VLES` that contained `NOP`s at `LA` or `LA-1`. This would have destroyed `LPMARKx`.

For More Information: www.freescale.com

Table B.3 Assembler Warnings (*continued*)

Warning	Explanation
`Ignoring -p option. Please use -arch` `Ignoring -r option. Please use -arch`	Command line included obsolete `-p` or `-r` option instead of current `-arch` option.
`Invalid lpmark due to empty packet(s)`	Attempt to set an `LPMARKx` on an empty packet, possibly a loop body that contained only directives.
`Label field ignored`	Label was not valid for the directive; the assembler did not store the label value in the symbol table.
`LDB option with no listing file specified; using source file`	`LDB` option lacked a listing file; the assembler used the source file.
`Load location counter overflow` `Load location counter underflow`	Load location counter value went outside its maximum or minimum bound. The assembler wrapped the counter value around and continued.
`Macro expansion not active`	Attempt to use the `@ARG()` or `@CNT()` function without a previous macro call.
`Macro name is the same as existing assembler directive` `Macro name is the same as existing assembler mnemonic`	Proposed macro name duplicates a directive or mnemonic; use a different name. Alternatively, use the `RDIRECT` directive to remove the directive/mnemonic from the assembler lookup tables.
`More than one overlay section in a module may result in corrupted debug information`	Multiple overlay sections in modules are not compatible with `-noovldbg` mode.
`Number of macro expansion arguments is greater than definition` `Number of macro expansion arguments is less than definition`	Argument-number discrepancy between the macro definition and the macro call.
`Number of long loops greater than number of dosetup instructions`	Possible missing `DOSETUP` instruction; the number of `DOSETUP` instructions must match the number of long loops.
`Opening bracket following another opening bracket or there is an unclosed bracket` `Opening bracket unclosed at end of file`	Missing closing bracket.

Table B.3 Assembler Warnings (*continued*)

Warning	Explanation
`Options for both absolute and C mode specified - C mode ignored`	Command line inappropriately had both `-a` and `-c` options. Assembler ignored the `-c` option.
`Options for both debug and strip specified - strip ignored`	Command line inappropriately had both `-g` and `-z` options. Assembler ignored the `-z` option.
`Option not implemented`	ELF-mode implementation does not include `AL`, *`CONST`*, `CRE`, `DBL`, `EM`, `GS`, `LB`, `LBX`, `NDE`, `NS`, `RP`, `SCL`, `SCO`, `S`, `XLL`, or `XROPT` options.
`Page directive with no arguments ignored with page length of zero`	Previous `PAGE` directive specified page length zero, so the assembler ignored this `PAGE` directive with no arguments. (Otherwise, the `PAGE` directive with no arguments is a listing-output form feed.)
`PRCTL directive ignored - no explicit listing file.`	Command line lacked the `-l` option, which is essential for the `PRCTL` directive.
`Redefinition of symbol`	`DEFINE` directive uses a symbol redefined without an intervening `UNDEF` directive. The assembler replaces the previous definition with the new one.
`Runtime location counter overflow` `Runtime location counter underflow`	Runtime location counter value went outside its maximum or minimum bound. The assembler wrapped the counter value around and continued.
`Storage block size not a power of 2`	An expression in a `DSR` directive did not evaluate to a power of 2. As the assembler relies on the `DSR` directive to allocate `FFT` buffers for reverse-carry operations, such buffers may not have appropriate sizes.
`String truncated in expression evaluation`	Assembler uses only the first four characters of string constants during expression evaluation. (`DC` directive arguments are exceptions.)
`Strip not valid in relocatable mode - ignored`	Command line include `-z` option, but lacked the corresponding `-a` option; the assembler ignored `-z`.
`Symbol alignment greater than section alignment`	`SIZE` directive specifies a symbol, but section that defines that symbol specifies smaller alignment than does the `SIZE` directive.

Table B.3 Assembler Warnings (*continued*)

Warning	Explanation
`Symbol is not properly aligned`	`SIZE` directive specifies a symbol, and specifies alignment for that symbol. But the address of the symbol's definition does not match the alignment of the `SIZE` directive.
`Unresolved external reference`	Source file did not include the definition of the specified symbol.
`Valid options for -arch: sc110,sc120,sc140,sc140e,msc8101,msc8102,pog, rainbow, starlite, sc1200, sc1400`	Invalid argument for command-line `-arch` option.
`Valid -s arguments: all,strict,none or individual restriction list`	Invalid argument for command-line `-s` option.

Table B.4 Assembler Error Messages

Message	Explanation
`Align size must be a power of 2 Align size must be greater than zero`	Section alignment value must be greater than zero and a power of 2.
`An empty packet can't be FALIGNed`	Attempt to falign a `VLES` that did not contain any instructions.
`Argument outside function domain`	Inappropriate argument to a transcendental built-in function.
`Assembler directive or mnemonic not found`	Assembler did not recognize an `RDIRECT`-directive argument as directive or mnemonic.
`At least two instructions have the same pc`	Command line `-k` option enabled data analysis, but two or more instructions from different sections had the same PC value. This prevents the assembler from determining a `jmp` destination.
`Base argument larger than machine word size`	Base-parameter value for a `@FLD()` function cannot fit into the target-machine word.
`Binary constant expected`	A character other than ASCII '0' or '1' followed the % binary-constant delimiter or appeared in an expression where a binary character should have been.

Assembler Messages

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation
`Cannot alloc data buffer for section`	Out of memory space, out of disk space, or an internal error.

`Cannot create ELF .bss section`
`Cannot create ELF header`
`Cannot create ELF section`
	Out of memory space, out of disk space, or an internal error.

`Cannot create ELF string table`
`Cannot create ELF symbol table`
`Cannot create ELF section header string table`
	Out of memory space, out of disk space, or an internal error.

`Cannot find loopstart for CONT(D)`	Command line `-k` option enabled data analysis, but the assembler could not find a `loopstart` connected to `CONT(D)`.
`Cannot nest section inside itself`	A section included a `SECTION` directive that specified the section's own name.
`Cannot nest symbol definitions`	A `.DEF-.ENDEF` directive pair included another `.DEF` directive.
`Cannot open include file`	Assembler could not find the specified `INCLUDE` file, or you have exceeded the operating-system limit on open files.
`Cannot parse address in org directive`	Assembler could not evaluate the address
`Cannot process indirect jumps with no special info`	Command line `-k` option enabled data analysis, but the assembler did not know the jump target. This prevented `CFG` creation.
`Cannot process indirect calls with no special info`	Command line `-k` option enabled data analysis, but the assembler did not know the subroutine-call target. This prevented data analysis.
`Cannot terminate ELF`	Out of memory space, out of disk space, or an internal error.
`Can't create node`	Assembler could not allocate memory.
`Can't find an unused source register for CLR`	All D registers were used four times.
`CMPHIA/CMPGTA/CMPEQA cannot compare the same register`	Both sources were the same register for a compare instruction.

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation
`CONST option must be used before any label`	Label appeared in the source file before the `CONST` option. (This option prevents exporting `EQU` symbols to the object file.)
`Count must be an integer value`	`DUP`-directive argument expression did not evaluate to an integer value.
`Counter value too large, max is 99999`	Counter value exceeded the 99999 maximum.
`CRE option must be used before any label`	Label appeared in the source file before the `CRE` option. (This option enables the assembler to append cross-reference data to all applicable symbol-table entries.
`Data allocation exceeds buffer size`	A `BUFFER – ENDBUF` directive sequence enclosed a data allocation that exceeds the size `BUFFER` specified.
`Decimal constant expected`	A character other than ASCII 0 through 9 followed the ‘ decimal-constant delimiter or appeared in an expression where a decimal character should have been.
`DEFINE symbol must be a global symbol name`	Attempt to use a local label (a symbol beginning with an underscore) as a symbol for the `DEFINE` directive.
`Divide by zero`	Expression evaluator detected a dive-by-zero situation.
`Dummy argument not found`	Assembler could not find the `@ARG()`-function dummy argument name in the macro dummy argument list.
`ELF buffer write failed` `ELF file update failed`	Operating system I/O error.
`ELF stack is empty`	Attempt to nest a section within itself.
`ELF symbol table error` `ELF symbol table insert failed`	Assembler could not allocate memory.
`.ELSE without associated IF statement`	No IF directive preceded an `ELSE` directive.
`ENDBUF without associated BUFFER directive`	No `BUFFER` directive preceded an `ENDBUF` directive.
`.ENDEF without associated .DEF directive`	No `.DEF` source-level debug directive preceded an `.ENDEF` directive.

Assembler Messages

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation
`.ENDF without associated .FOR statement`	No `.FOR` directive preceded an `.ENDF` directive.
`.ENDW without associated .WHILE statement`	No `.WHILE` directive preceded an `.ENDW` directive.
`ENDIF without associated IF directive`	No `IF` directive preceded an `ENDIF` directive.
`ENDM without associated MACRO directive`	No `MACRO` directive preceded an `ENDM` directive.
`ENDSEC without associated SECTION directive`	No `SECTION` directive preceded an `ENDSEC` directive.
`EQU requires label`	`EQU` directive lacked a label to associate with the equated expression.
`Error in mnemonic table`	No `MACRO` directive preceded an `EXITM` directive.
`Errors detected. Data analysis cannot be performed`	Command line `-k` option enabled data analysis, but the assembler encountered errors severe enough to prevent data analysis.
`EXITM without associated MACRO directive`	No `MACRO` directive preceded an `EXITM` directive.
`Expression cannot have a negative value`	Negative expression argument for `PAGE` or other such directive that does not allow negative arguments.
`Expression contains forward references`	Undefined expression argument for `IF`, `EQU`, `SET`, or other such directive that does not allow forward references.
`Expression must be greater than zero`	Zero expression argument for `BSC` or other such directive that does not allow zero arguments.
`Expression result must be absolute`	Non-absolute argument or operand for a directive or usage that requires an absolute value.
`Expression result must be integer`	Non-integer argument or operand for a directive or usage that requires an integer value.
`Expression result too large`	Expression evaluated to a value greater than the acceptable range, possibly because the result exceeded the target-DSP native word size.

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation
`Expression too long`	REL expression exceeded the 1024-character limit.
`External reference not allowed in expression`	Reference to an external symbol in a byte or integer expression that does not allow such references.
`External reference not allowed in function`	Reference to an external symbol in an argument to a built-in function.
`Extra characters beyond expression`	Expression evaluator found extra characters after the end of a valid expression, possibly due to unbalanced parentheses.
`Extra characters following string`	In the source line, unexpected characters followed the end-of-string delimiter.
`Extra characters following symbol name`	Symbol name contained a non-alphabetic character other that an underscore.
`Extra characters in function argument or missing ')' for function`	Mismatched parentheses or wrong number of parameters in a function call.
`Extra characters in operand field`	Too many operands for a `PAGE` directive.
`Extra fields ignored`	Assembler directive specified extra fields.
`EXTRACT: Offset + width > 40 bits`	Violation of the 40-bit limit; specific only to the sc100 processor.
`Floating point constant expected`	A character other than ASCII 0 through 9, e, E, or '.' appeared in an expression where a floating-point value should have been.
`Floating point not allowed in relative expression`	Floating point value in a relative expression — not appropriate, as the assembler uses relative expressions for address computations.
`Forcing not specified`	`FORCE` directive lacked the type of forcing operand.
`Forward reference not allowed for expression`	Given context did not permit a forward or external reference.
`Function result out of range`	Transcendental function computed a result too large for the host machine to represent.

Assembler Messages

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation
GL option must be used before any section	Assembler encountered an explicit section before encountering the GL option; this option makes sure all section symbols are global.
GLOBAL without preceding SECTION directive	Assembler encountered a GLOBAL directive outside any previously established section.
GS option must be used before any section	Assembler encountered an explicit section before encountering the GS option; this option lets the assembler use appropriate counters for section relocation
GSET may not be used to redefine a local symbol	In ELF mode, the GSET directive cannot redefine local symbols.
Hex constant expected	A $ character appeared without a hexadecimal value.
IC option must be used before any symbol, section, or macro definition	IC option was not activated when the assembler encountered a symbol, section, or macro definition. This option keeps the assembler consistent regarding label names in the symbol table.
IDENT directive must contain revision number IDENT directive must contain version number	IDENT directive lacked the revision or version number; both are required.
Illegal directive in buffer declaration	A BUFFER — ENDBUF directive sequence enclosed a directive not valid in that context. Examples include such buffer directives as DSM or DSR section directives, and any directive such as MODE or ORG that changes the location counter designation.
Illegal directive inside .DEF-.ENDEF declaration	.FILE or other directive that made no sense was inside a .DEF—.ENDEF declaration.
Illegal directive inside DO loop	A DO loop enclosed a directive not valid in that context. Examples include such buffer directives as DSM or DSR section directives, and any directive such as MODE or ORG that changes the location counter designation.
Illegal directive outside .DEF-.ENDEF declaration	.VAL or other directive that made no sense was outside a .DEF—.ENDEF declaration.
Illegal escape character specified	Escape character was not \n, \t, \f, \a, \r, \v, \\, or ".

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation
`Illegal function argument`	`@LCV()` or another built-in assembler function received an invalid argument.
`Illegal memory counter specified` `Illegal memory map character`	Specified memory counter or memory-map character was not compatible with P memory space. (Only P memory space is valid for StarCore.)
`Illegal memory space character` `Illegal memory space specified`	Specified memory space or memory-space character was not compatible with P memory space. (Only P memory space is valid for StarCore.)
`Illegal local label`	ELF-mode label did not start with %.
`Illegal operator for floating point element`	Source code specified a bitwise operator for a floating-point value.
`Illegal option`	Undefined `-o` option.
`Increment value cannot be zero`	DUPF directive increment parameter value must be greater than zero.
`Initial debug directive must be .FILE`	In a source file that contained debug directives, being assembled withe the `-G` option, the `.FILE` directive was not the first source-level debug directive in the input stream.
`INSERT: Offset + width > 40 bits`	Violation of the 40-bit limit; specific only to the sc100 processor.
`Instruction cannot have ift, iff, ifa in one packet`	Violation of an sc100 processor restriction.
`Instruction could not be reordered for MUX`	Violation of an sc100 processor restriction.
`Invalid buffer type`	`BADDR` or `BUFFER` directive specified a type that was not M (modulo) or R (reverse-carry).
`Invalid dummy argument name`	Macro dummy argument name was a local symbol, possibly beginning with the % character.
`Invalid function name`	Assembler did not recognize the name after the @ character.
`Invalid label field width specified`	`LSTCOL`-directive argument did not allow enough room on the listing line for the remaining fields.

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation
`Invalid macro name`	Macro name was a local symbol, possibly beginning with the % character.
`Invalid memory space attribute`	Memory space attribute was not the letter P.
`Invalid mode`	`MODE` directive specified a mode that was not RELATIVE or ABSOLUTE.

`Invalid opcode field width specified`
`Invalid opcode 2 field width specified`
`Invalid operand field width specified`
	`LSTCOL`-directive argument did not allow enough room on the listing line for the remaining fields.

`Invalid page length specified`	`PAGE` directive specified fewer than 10 lines or more than 255 lines.
`Invalid page width specified`	`PAGE` directive specified fewer than 1 column or more than 255 columns.
`Invalid radix expression`	`RADIX` directive expression did not evaluate to 2, 8, 10, or 16.
`Invalid relative expression`	Code specified an operation other than addition or subtraction for relative-expression terms, or the terms had the same sign.
`Invalid section name`	Section name was a local symbol, possibly beginning with the % character.
`Invalid shift amount`	A shift expression evaluated to less than zero, or to more than the target-DSP maximum address.
`Invalid storage class`	Assembler did not know the storage class for a source-level debug symbol declaration.
`Invalid tab stops specified`	`TAB` directive argument was out of range.

`Label specified does not exist`
`Label specified in CONT(D) is not defined`
`Label specified in SKIPLS is not defined`
	Command line `-k` option enabled data analysis, but the assembler could not find the specified label.

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation

`LB option must be used before any code or data generation`
`LBX option must be used before any code or data generation`
`LDB option must be used before any code or data generation`
	Assembler encountered code or data before encountering the `LB`, `LBX`, or `LDB` option.

`Left margin exceeds page width`	`PAGE` directive left-margin value was blank, exceeding the default or specified page width value.
`Line too long`	Source statement was longer than 4000 characters, possibly due to a continuation line.
`LOC option must be used before any local label`	`LOC` option did not appear before local labels; this option enables the assembler to synchronize local label lists.
`LOCAL directive not valid in global section` `LOCAL not allowed in global section`	A local directive was not specified inside a section.
`Local symbol names cannot be used with GLOBAL`	The directive does not allow % labels.
`LOCAL without preceding SECTION directive`	A local directive was not specified inside a section.
`Loopend without loopstart`	Loopend directive lacked the beginning loopstart directive.
`Loop stack error, possible loopstart-loopend mismatch`	Invalid combinations of loopstart-loopend directives.
`Loopstart without loopend`	Loopstart directive lacked the concluding loopend directive.
`Macro cannot be redefined`	Attempt to use a macro name as the label for a second macro definition in the same source file. This would be possible only if the macro were defined and used within a declared section.
`Macro not defined`	The assembler could not find the macro name in the macro lookup table.
`Macro value substitution failed`	The evaluation of a macro argument expression failed.

Assembler Messages

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation
`Memory bounds greater than maximum address`	Invalid bounds argument in a `LOMEM` or `HIMEM` directive.
`Memory counter designator value too large`	`ORG`-directive integer counter designator value exceeded 65535.
Memory space must be P or NONE	The assembler encountered an `END` directive when the runtime memory space was neither P nor N.
`Missing "(" for function`	An assembler-built function lacked any argument enclosed in parentheses; all such functions need at least one.
`Missing ")" in expression`	Unbalanced parentheses in an expression.
`Missing argument`	The assembler could not find the argument for a `DUPA` or `DUPC` directive.
`Missing definition string`	`DEFINE` directive lacked its substitution string.
`Missing delimiter in substring`	A substring construct lacked its closing bracket.
`Missing directive name`	`RDIRECT` directive lacked a directive argument.
`Missing expression`	An expression should have followed a comma.
`Missing filename`	`INCLUDE` directive lacked a filename argument.
`Missing directive name`	`RDIRECT` directive lacked a directive argument.
`Missing label name`	`TYPE` directive lacked a label argument.
`Missing macro name`	`MACRO` directive lacked a label, or a `PMACRO` directive lacked the macro name.
`Missing option`	`OPT` directive lacked an argument.
`Missing or mismatched quote`	Parsing routines did not find the appropriate single- or double-quote character.
`Missing pathname`	`MACLIB` directive lacked a pathname argument.
`Missing quote`	String parsing routines did not find the appropriate single- or double-quote character.
`Missing section name`	`SECTION` directive lacked a section-name argument.

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation
`Missing string after concatenation operator`	A quoted string did not follow the ++ operator.
`Missing symbol name`	`DEFINE` directive first argument was not a symbol name.
`Missing type definition`	`TYPE` directive did not refer to a label.
`Mode not specified`	Neither RELATIVE nor ABSOLUTE followed the `MODE` directive.
`More than one pseudo on a line or pseudo mixed with instructions`	Only one pseudo may appear on a line; the same list must not mix pseudos and instructions.
`MU option must be used before any code or data generation`	Assembler encountered a data allocation directive (`BSC`, `DC`, `DS`, `DSR`) or an instruction before encountering the MU option.
`No instructions in enclosing blocks`	No instructions were inside a bracket set.
`NOGS option must be used before any section`	Assembler encountered an explicit section before encountering the `NOGS` option; if you use this option, it must precede the `SECTION` directive.
`One or more of the specified targets do not exist`	Command line `-k` option enabled data analysis, but at least one indirect-jump `Lint_info` target did not exist.
`Operation not allowed with address term`	An expression involving addresses or relative terms included an operation other than addition or subtraction.
`Page length too small for specified top and bottom margins`	In a `PAGE` directive, the top-and-bottom margin value sum exceeded the value (page length - 10).
`Page length too small to allow default bottom margin`	`PAGE`-directive bottom margin value exceeded the page length value.
`PC-relative address too large to use short`	PC-relative offset being forced short was too large to fit into the instruction word.
`PC-relative addressing mode not allowed`	The instruction does not allow PC-relative addressing mode. (Bit-manipulation instructions and some jump-type instructions have this restriction.)

Assembler Messages

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation
`Phasing error`	Value associated with a symbol changed between assembly passes 1 and 2; this error can occur spontaneously in conjunction with other errors. • Using the checksumming instruction `@CHK()` with the `EQU` directive could be the problem. Forward referencing could leave instruction coding incomplete after the first pass, so that the checksum value changes between passes. To correct this problem, use the `SET` directive to assign the checksum value to a symbol. • A phasing error without any other errors may represent an internal error that you should report to StarCore.
`Possible invalid white space between operands or arguments`	Fields that should not contain operands or values were not empty.
`Problem installing label`	Incorrect symbol name.
`RDIRECT directive not allowed in section`	`RDIRECT` directive was inside a section declared with a `SECTION` directive. This is inappropriate, as `RDIRECT`'s effect is global. Move `RDIRECT` outside the section.
`Redefinition would overflow line`	Substitution, according to the `DEFINE`-directive substitution string, would cause an overflow of the current source line.
`Reference outside of current buffer block` `Reference outside of current overlay block`	Assembler encountered a reference to an underscore local label that fell outside the current buffer or overlay definition.
`Relative equate must be in same section`	Definition of an `EQU` directive with a relative expression operand was not in the section associated with the operand expression.
`Relative equate must share the same counter`	Invalid operation between two symbols, as their definitions involved different counters.
`Relative expression must be integer`	Context of a relative expression required an integer expression.
`Relative expression not allowed`	A relative expression appeared as the argument to an assembler built-in function.

For More Information: www.freescale.com

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation
`Relative SET must be in same section`	Definition of a `SET` directive with a relative expression operand was not in the section associated with the operand expression.
`Relative terms from different sections not allowed`	An expression contained relative terms defined in different sections. (Definitions must be in the same section, as that section's location in memory determines the relationship between the terms.)
`Reserved name used for symbol name`	Proposed label, operand, or directive argument duplicated a reserved `DSP` register name.
`Runtime space must be P`	In COFF mode, instructions must appear in P memory space.
`SCO option must be used before any label`	In COFF mode, the `SCO` option must appear before any label definition.
`SET requires label`	`SET` directive lacked a label, required to associate the directive argument with a symbol name.
`SET symbol names cannot be used with GLOBAL`	Attempt to use `GLOBAL` to export a symbol from a section, when the `SET` directive had defined the symbol
`SIZE directive should be used after defining symbol type`	`SIZE` directive did not use a defined symbol type.
`SP cannot be used as an operand in DECEQA/DECGEA` `SP cannot be used as an operand of IMCA/DECA`	Violation of specific-processor restriction.
`Start argument greater than machine word size`	`@FLD()` function start parameter value did not fit in the target-machine word.
`Start position greater than source string size`	`@POS()` function start parameter value exceeded the total length of the source string argument.
`Storage block size must be greater than zero`	Size was too small for a buffer associated with the `DSM`, `DSR`, `BSM`, `BSB`, or other buffer directive.
`Storage block size out of range`	Size was too large to be allocated for a buffer associated with the `DSM`, `DSR`, `BSM`, `BSB`, or other buffer directive.

Assembler Messages

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation
`Storage block too large`	As the assembler attempted to allocate storage, via a `DSR` directive, the runtime location counter overflowed. (The assembler automatically advances the program counter to the next valid base address, given the size of the modulo or reverse-carry buffer. This overflow occurs if the sum of the `DSR`-directive expression and the runtime location counter value exceeds available memory in the current memory space.)
`SVO option must be used before any code or data generation`	Assembler encountered a data allocation directive (`BSC`, `DC`, `DS`, `DSM`, `DSR`) or an instruction before encountering the `SVO` option.
`Symbol already defined as GLOBAL`	`GLOBAL` directive defined a symbol that a previous `GLOBAL` directive already defined.
`Symbol already used as SET symbol`	`SET` directive already defined the label; only another `SET` directive can redefine that label.
`Symbol cannot be set to new value`	A directive other than `SET` defined the label; such labels cannot be redefined.
`Symbol defined in current section before GLOBAL directive`	In a section, the assembler encountered a symbol definition before the `GLOBAL` directive. The `GLOBAL` directive should follow the `SECTION` directive immediately.
`Symbol name too long.`	The symbol name exceeded the 512-character limit.
`Symbol not previously defined`	`UNDEF` directive specified a symbol not previously defined by a `DEFINE` directive.
`Symbol redefined`	Code already used the symbol as a label in a previous context.
`Symbol tag mismatch`	Assembler could not find a matching tag reference for a tagged symbol table entry.
`Symbol undefined on pass 2`	Source program never defined the symbol used as an operand or directive argument.
`Symbols must start with alphabetic character`	Symbol name began with a or other inappropriate character. First character must be a letter, percent sign, or underscore.

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation
SYMOBJ symbol must be a global symbol name	A SYMOBJ-directive argument begins with an underscore.
Syntax error - expected '):'	ORG directive using numeric counter designations lacked the parenthesis/colon pair that separates the load or runtime address from the memory space, counter, or mapping characters.
Syntax error - expected ':'	ORG directive lacked the colon that separates the load or runtime address from the memory space, counter, or mapping characters.
Syntax error - expected '>'	Argument of a non-local INCLUDE directive lacked the closing angle bracket.
Syntax error - expected comma	Operand list of an instruction or directive lacked a separating comma.
Syntax error - expected keyword BY	FOR directive lacked its associated BY directive.
Syntax error - expected keyword DO	FOR directive lacked its associated DO directive.
Syntax error - expected keyword TO or DOWNTO	FOR directive lacked its associated TO or DOWNTO directive.
Syntax error - expected quote	Start of a quoted string should have appeared at the location in code.
Syntax error - extra characters	Extra characters appeared after an instruction or directive operand.
Syntax error - missing address mode specifier	Unspecified instruction operand.
Syntax error - invalid assignment operator	FOR directive lacked its right term.
Syntax error - invalid compound operator	Assembler did not find an AND, OR, THEN, or DO operator where appropriate.
Syntax error - invalid conditional operator	Assembler did not find the < or > operator where appropriate.
Syntax error - invalid expression format	Invalid optional BREAK expression.

Assembler Messages

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation
`Syntax error - invalid statement terminator`	Assembler did not find the appropriate `THEN`, `ENDW`, or `UNTIL` directive for the corresponding `IF`, `WHILE`, or `REPEAT` directive.
`Syntax error in directive name list`	`RDIRECT` directive lacked its appropriate comma.
`Syntax error in dummy argument list`	In a macro definition, a non-comma character separated dummy arguments, or an underscore character appeared.
`Syntax error in macro argument list`	In a macro call, a non-comma character separated arguments, or extra characters appeared.
`Syntax error in macro name list`	In a `PMACRO` directive name list, a non-comma character separated arguments.
`Syntax error in symbol name list`	Comma missing where appropriate.
`Tag name not found`	Assembler could not find a matching tag name for the current source-level debug structure or union declaration.
`The specified target does not exist`	Command line `-k` option enabled data analysis, but this module does not define the destination of the direct jump instruction. This prevents data analysis.
`The string should be surrounded by "`	`NOTE` directive string operand lacks its enclosing double quote characters.
`Too many fields specified for instruction`	An instruction field should have been empty, but contained non-comment data.
`Two dummy arguments are the same`	In a macro definition, two dummy arguments have the same name.
`UNDEF symbol must be a global symbol name`	`UNDEF`-directive argument was a local label.
`Unexpected end of file - missing COMMENT delimiter`	`COMMENT` directive lacked its second delimiter character.
`Unexpected end of file - missing ENDBUF`	Assembler encountered a `BUFFER` directive without the closing `ENDBUF` directive.
`Unexpected end of file - missing ENDIF`	Assembler encountered an `IF` directive without the closing `ENDIF` directive.

Table B.4 Assembler Error Messages (*continued*)

Message	Explanation
`Unexpected end of file - missing ENDM`	Assembler encountered a `MACRO` directive without the closing `ENDM` directive.
`Unexpected end of file - missing ENDSEC`	Assembler encountered a `SECTION` directive without the closing `ENDSEC` directive.
`Unknown math error`	A transcendental math function returned an error that the assembler could not classify as out of range or outside the function domain.
`Unrecognized mnemonic`	Assembler opcode field contained a symbol not a defined macro, instruction mnemonic, or a directive.
`.UNTIL without associated .REPEAT statement`	Missing `REPEAT` directive.
`Value argument larger than machine word size`	`@FLD()`-function value parameter value too large to fit into the target-machine word.
`Width argument greater than machine word size`	`@FLD()`-function width parameter value too large to fit into the target-machine word.
`XDEF without preceding SECTION directive`	Assembler encountered an `XDEF` directive before encountering a `SECTION` directive. (Pertains only to COFF mode; ELF mode does not support `XDEF`.)
`XLL option must be used before any local label`	In COFF mode, the `XLL` option must precede all local-label definitions.
`XR option must be used before any label`	Assembler encountered a label when the `XR` option was not active. This option lets the assembler make appropriate entries in the symbol table.
`XREF without preceding SECTION directive`	Assembler encountered an `XREF` directive before encountering a SECTION directive. (Pertains only to COFF mode; ELF mode does not support `XREF`.)

Assembler Messages

Table B.5 Assembler Fatal Error Messages

Message	Explanation
`Attempt to store external reference data in absolute mode`	Storing external reference data is valid only in COFF mode, not in absolute mode.

`Cannot seek to start of line number entries`
`Cannot seek to start of object data`
`Cannot seek to start of object file`
`Cannot seek to start of relocation entries`
`Cannot seek to start of section headers`
`Cannot seek to start of string table`
`Cannot seek to start of symbol table`
	Operating system I/O error prevented the assembler from positioning the item correctly in the output object file.

`Cannot write file header to object file`
`Cannot write line number entries to object file`
`Cannot write optional header to object file`
`Cannot write relocation entries to object file`
`Cannot write section headers to object file`
`Cannot write string table to object file`
`Cannot write symbols to object file`
	Operating system I/O error prevented the assembler from writing data element to the output object file.

For More Information: www.freescale.com

Table B.5 Assembler Fatal Error Messages (*continued*)

Message	Explanation

`Cannot write control string to listing file`
`Cannot write form feed to listing file`
`Cannot write left margin to listing file`
`Cannot write new line to listing file`
`Cannot write new page to listing file`
`Cannot write page header to listing file`
`Cannot write string to listing file`
	Operating system I/O error prevented the assembler from writing data element to the output listing file.

`Can't find an unused source register for CLR`	All registers have been used four times.
`Debug information for multiple file assembling is inaccurate`	Assembler can assemble only one file.
`Expression operator failure`	Failure of the expression operator lookup. You should report this serious internal error to StarCore.
`Falign is not allowed in a VLES`	`FALIGN` directive appeared inside a `VLES`.
`I/O error writing data word to object file`	Operating system I/O error prevented the assembler from writing data to the output object file.
`Loopend is not allowed in a VLES`	`LOOPEND` directive appeared inside a `VLES`.
`Loopstart is not allowed in a VLES`	`LOOPSTART` directive appeared inside a `VLES`.
`Nesting error - possible endsec without its corresponding section`	Assembler found an `ENDSEC` directive without its corresponding `SECTION` directive.
`Out of memory`	Assembler could not allocate memory.
`Too many lines in source file`	A source file exceeded the maximum number of lines: 2 to the 32nd power.
`Too many sections in module`	A module exceeded the maximum number of sections: 2 to the 16th power.

For More Information: www.freescale.com

Index

Symbols

" (delimit macro string) directive 96, 97
" (expand define quoted string) directive 97
(use immediate addressing) directive 100
#< (force immediate short addressing) directive 101
#> (force immediate long addressing) directive 101, 102
% (substitute macro hex value) directive 95
* (substitute location counter) directive 98
++ (concatenate strings) directive 98
; (start comment) directive 92
;; (start unreported comment) directive 92
< (force short addressing) directive 99, 100
> (force long addressing) directive 100
? (substitute macro value) directive 94, 95
@ (start function) directive 97, 98
[] (delimit substring) directive 98, 99
[] (group instructions) directive 99
\ (concatenate macro argument) directive 93, 94
\ (continue line) directive 93
^ (override macro local label) directive 96

A

ABS (absolute value) function 53
absolute expressions 45
absolute mode, multi-programmer environment 85
absolute value (ABS) function 53
ACS (arc cosine) function 53, 54
additional directories, searching 20, 21
address assignment 78, 79
advance/size page (PAGE) directive 140, 141
ALIGN (align location counter) directive 102
align location counter (ALIGN) directive 102
align with fetch-set (FALIGN) directive 123, 124
allocate bit-reverse buffer (BSB) directive 103, 104
allocate block storage for modulo (BSM) directive 105, 106
allocate constant storage block (BSC) directive 104, 105
analysis, data 28–32
arc cosine (ACS) function 53, 54
arc sine (ASN) function 54
arc tangent (AT2) function 55
arc tangent (ATN) function 55
ARG (macro argument) function 54
argument file, reading from 17, 18
ASCII character codes 161, 162
ASN (arc sine) function 54
assembler
    error messages 169–185
    fatal error messages 186, 187
    processing 35, 36
    SC100 13–44
    starting 13, 14
    warnings 166–169
assembler mode functions 52, 53
assembly control directives 89
assignment, address 78, 79
AT2 (arc tangent) function 55
ATN (arc tangent) function 55
attributes, memory space 45, 46

B

BADDR (set buffer address) directive 102, 103
BIGENDIAN (endian mode check) function 55, 56
BSB (allocate bit-reverse buffer) directive 103, 104
BSC (allocate constant storage block) directive 104, 105
BSM (allocate block storage for modulo) directive 105, 106
BUFFER (start buffer) directive 106, 107

C

calling macros 156, 157
CCC (cumulative cycle count) function 56
ceiling (CEL) function 56

For More Information: www.freescale.com

CEL (ceiling) function 56
character codes, ASCII 161, 162
characters, significant in directives 87, 88
checking programming rules 23–28
CHK (instruction/data checksum) function 57
CNT (macro argument count) function 57
code examples 32, 33
COH (hyperbolic cosine) function 57, 58
command-line
error messages 164, 165
options 14–35
warnings 163, 164
COMMENT (start comment lines) directive 107, 108
comment field, source statements 38
compare strings (SCP) function 70
concatenate macro argument (\) directive 93, 94
concatenate strings (++) directive 98
concatenate to double word (LNG) function 65
conditional assembly 158, 159
conditional assembly directives 91
constants
in expressions 46, 47
numeric 46, 47
string 47
continue line (\) directive 93
controlling assembler messages 20
conversion functions 52
convert floating point to fractional (FRC) function 62
convert floating point to integer (CVI) function 59
convert floating point to long fractional (LFR) function 64, 65
convert fractional to floating point (UNF) function 73
convert integer to floating point (CVF) function 59
convert long fractional to floating point (LUN) function 66
convert memory space (CVS) function 59, 60
COS (cosine) function 58
cosine (COS) function 58
CTR (location counter number) function 58
cumulative cycle count (CCC) function 56
CVF (convert integer to floating point) function 59
CVI (convert floating point to integer) function 59
CVS (convert memory space) function 59, 60

D

data analysis 28–32
limitations 33, 34
terms 33
data definition directives 90
DC (define constant) directive 108, 109
DCB (define constant byte) directive 109, 110
DCL (define constant long) directive 110
declare global section symbol (GLOBAL) directive 124, 125
DEF (defined symbol) function 60
DEFINE (define substitution string) directive 110, 111
define constant (DC) directive 108, 109
define constant byte (DCB) directive 109, 110
define constant long (DCL) directive 110
define external symbol (XDEF) directive 153
define macro (MACRO) directive 130, 131
define reverse-carry storage (DSR) directive 113
define storage (DS) directive 111, 112
define storage for modulo (DSM) directive 112, 113
define substitution string (DEFINE) directive 110, 111
define symbols in sections 78
defined symbol (DEF) function 60
defining
macros 155, 156
substitution strings 21
delimit macro string (") directive 96, 97
delimit substring ([]) directive 98, 99
development flow 9–11
directive list 88–91
directives 87–154
" (delimit macro string) 96, 97
" (expand define quoted string) 97
(use immediate addressing) 100

#< (force immediate short addressing) 101
#> (force immediate long addressing) 101, 102
% (substitute macro hex value) 95
* (substitute location counter) 98
++ (concatenate strings) 98
; (start comment) 92
;; (start unreported comment) 92
< (force short addressing) 99, 100
> (force long addressing) 100
? (substitute macro value) 94, 95
@ (start function) 97, 98
[] (delimit substring) 98, 99
[] (group instructions) 99
\ (concatenate macro argument) 93, 94
\ (continue line) 93
^ (override macro local label) 96
ALIGN (align location counter) 102
assembly control 89
BADDR (set buffer address) 102, 103
BSB (allocate bit-reverse buffer) 103, 104
BSC (allocate constant storage block) 104, 105
BSM (allocate block storage for modulo) 105, 106
BUFFER (start buffer) 106, 107
COMMENT (start comment lines) 107, 108
conditional assembly 91
data definition 90
DC (define constant) 108, 109
DCB (define constant byte) 109, 110
DCL (define constant long) 110
DEFINE (define substitution string) 110, 111
descriptions 91–154
DS (define storage) 111, 112
DSM (define storage for modulo) 112, 113
DSR (define reverse-carry storage) 113
DUP (duplicate source lines) 114
DUPA (duplicate sequence with arguments) 115
DUPC (duplicate sequence with characters) 116
DUPF (duplicate sequence in loop) 117
ELSE (start alternative conditional assembly) 118
END (end of source program) 118, 119
ENDBUF (end buffer) 119
ENDIF (end conditional assembly) 119, 120
ENDM (end macro definition) 120
ENDSEC (end section) 121
EQU (equate symbol to value) 121, 122
EXITM (exit macro) 122
FAIL (issue programmer error message) 122, 123
FALIGN (align with fetch-set) 123, 124
GLOBAL (declare global section symbol) 124, 125
GSET (set global symbol to value) 125
HIMEM (set high memory bounds) 125, 126
IF (start conditional assembly) 126, 127
INCLUDE (include secondary file) 127, 128
LIST (list assembly) 128, 129
listing control 90
LOMEM (set low memory bounds) 129
MACLIB (specify macro library) 129, 130
MACRO (define macro) 130, 131
macros 91
MSG (issue programmer message) 131, 132
NOLIST (stop assembly listing) 132
NOTE (include note) 132, 133
OPT (set assembler options) 133–139
options 90
ORG (initialize memory space) 139, 140
PAGE (advance/size page) 140, 141
PMACRO (purge macro definition) 141, 142
PRCTL (send control string to printer) 142
RADIX (set constant radix) 143
SECFLAGS (set ELF section flags) 143, 144
SECTION (start section) 144–147
SECTYPE (set ELF section type) 147, 148
SET (set symbol to value) 148
significant characters 87, 88
SIZE (set symbol size) 149
STITLE (initialize program subtitle) 149, 150

For More Information: www.freescale.com

storage allocation 90
symbol definition 89
TITLE (initialize program title) 150
TYPE (set symbol type) 151
UNDEF (undefine define symbol) 151, 152
WARN (issue programmer warning) 152, 153
XDEF (define external symbol) 153
XREF (reference external symbol) 154
directories, additional, searching 20, 21
DS (define storage) directive 111, 112
DSM (define storage for modulo) directive 112, 113
DSR (define reverse-carry storage) directive 113
dummy argument operators in macros 157, 158
DUP (duplicate source lines) directive 114
DUPA (duplicate sequence with arguments) directive 115
DUPC (duplicate sequence with characters) directive 116
DUPF (duplicate sequence in loop) directive 117
duplicate sequence in loop (DUPF) directive 117
duplicate sequence with arguments (DUPA) directive 115
duplicate sequence with characters (DUPC) directive 116
duplicate source lines (DUP) directive 114

E

ELSE (start alternative conditional assembly) directive 118
END (end of source program) directive 118, 119
end buffer (ENDBUF) directive 119
end conditional assembly (ENDIF) directive 119, 120
end macro definition (ENDM) directive 120
end of source program (END) directive 118, 119
end section (ENDSEC) directive 121
ENDBUF (end buffer) directive 119
endian mode check (BIGENDIAN) function 55, 56
endian mode, specifying 22, 23
ENDIF (end conditional assembly) directive 119, 120
ENDM (end macro definition) directive 120
ENDSEC (end section) directive 121
environment variable, using 17
EQU (equate symbol to value) directive 121, 122
equate symbol to value (EQU) directive 121, 122
error messages
assembler 169–185
command-line 164, 165
example
multi-programmer environment 84–86
overlay 81–84
source listing 41–43
examples, code 32, 33
execution sets, source statements 38, 39
exit macro (EXITM) directive 122
EXITM (exit macro) directive 122
EXP (expression check) function 60, 61
expand define quoted string (") directive 97
expansions, macros 157
exponential function (XPN) function 73
expression check (EXP) function 60, 61
expressions 45–73
absolute and relative 45
constants 46, 47
internal representation 46
memory space attributes 45, 46

F

FAIL (issue programmer error message) directive 122, 123
FALIGN (align with fetch-set) directive 123, 124
fatal error messages, assembler 186, 187
final information, source listing 43, 44
FLD (shift and mask) function 61
floor (FLR) function 62
FLR (floor) function 62
force immediate long addressing (#>) directive 101, 102
force immediate short addressing (#<) directive 101
force long addressing (>) directive 100
force short addressing (<) directive 99, 100
fragmented sections 76, 77

For More Information: www.freescale.com

FRC (convert floating point to fractional) function 62
functions 50–73
ABS (absolute value) 53
ACS (arc cosine) 53, 54
ARG (macro argument) 54
ASN (arc sine) 54
assembler mode 52, 53
AT2 (arc tangent) 55
ATN (arc tangent) 55
BIGENDIAN (endian mode check) 55, 56
CCC (cumulative cycle count) 56
CEL (ceiling) 56
CHK (instruction/data checksum) 57
CNT (macro argument count) 57
COH (hyperbolic cosine) 57, 58
conversion 52
COS (cosine) 58
CTR (location counter number) 58
CVF (convert integer to floating point) 59
CVI (convert floating point to integer) 59
CVS (convert memory space) 59, 60
DEF (defined symbol) 60
EXP (expression check) 60, 61
FLD (shift and mask) 61
FLR (floor) 62
FRC (convert floating point to fractional) 62
INT (integer check) 62, 63
L10 (log base 10) 63
LCV (location counter value) 63, 64
LEN (string length) 64
LFR (convert floating point to long fractional) 64, 65
LNG (concatenate to double word) 65
LOG (natural logarithm) 65
LST (LIST directive flag value) 66
LUN (convert long fractional to floating point) 66
MAC (macro definition) 66, 67
macro 52
mathematical 51
MAX (maximum value) 67
MIN (minimum value) 67
MSP (memory space) 67, 68
MXP (memory expansion) 68
POS (position of substring) 68
POW (raise to a power) 69
REL (relative mode) 69
RND (random value) 69
RVB (reverse bits in field) 70
SCP (compare strings) 70
SGN (return sign) 70, 71
SIN (sine) 71
SNH (hyperbolic sine) 71
SQT (square root) 72
string 52
TAN (tangent) 72
TNH (hyperbolic tangent) 72, 73
UNF (convert fractional to floating point) 73
XPN (exponential function) 73

G

generating an object file 18
GLOBAL (declare global section symbol) directive 124, 125
group instructions ([]) directive 99
GSET (set global symbol to value) directive 125

H

HIMEM (set high memory bounds) directive 125, 126
hyperbolic cosine (COH) function 57, 58
hyperbolic sine (SNH) function 71
hyperbolic tangent (TNH) function 72, 73

I

IF (start conditional assembly) directive 126, 127
INCLUDE (include secondary file) directive 127, 128
include note (NOTE) directive 132, 133
include secondary file (INCLUDE) directive 127, 128
initialization file 34, 35
initialization file directives 34, 35
initialize memory space (ORG) directive 139, 140

For More Information: www.freescale.com

initialize program subtitle (STITLE) directive 149, 150
initialize program title (TITLE) directive 150
instruction/data checksum (CHK) function 57
INT (integer check) function 62, 63
integer check (INT) function 62, 63
internal expression representation 46
introduction 9–11
issue programmer error message (FAIL) directive 122, 123
issue programmer message (MSG) directive 131, 132
issue programmer warning (WARN) directive 152, 153

L

L10 (log base 10) function 63
label field, source statements 37
LCV (location counter value) function 63, 64
LEN (string length) function 64
LFR (convert floating point to long fractional) function 64, 65
libraries, macros 157
limitations, data analysis 33, 34
LIST (list assembly) directive 128, 129
list assembly (LIST) directive 128, 129
LIST directive flag value (LST) function 66
list, directive 88–91
listing control directives 90
listing, source 40–44
LNG (concatenate to double word) function 65
location counter number (CTR) function 58
location counter value (LCV) function 63, 64
LOG (natural logarithm) function 65
log base 10 (L10) function 63
LOMEM (set low memory bounds) directive 129
LST (LIST directive flag value) function 66
LUN (convert long fractional to floating point) function 66

M

MAC (macro definition) function 66, 67
MACLIB (specify macro library) directive 129, 130
MACRO (define macro) directive 130, 131
macro argument (ARG) function 54
macro argument count (CNT) function 57
macro definition (MAC) function 66, 67
macro functions 52
macros 155–158
  calling 156, 157
  defining 155, 156
  directives 91
  dummy argument operators 157, 158
  expansions 157
  in sections 78
  libraries 157
manager, overlay 80, 81
mathematical functions 51
MAX (maximum value) function 67
maximum value (MAX) function 67
memory expansion (MXP) function 68
memory space (MSP) function 67, 68
memory space attributes 45, 46
messages
  assembler error messages 169–185
  assembler fatal error messages 186, 187
  assembler warnings 166–169
  command-line error messages 164, 165
  command-line warnings 163, 164
  controlling 20
MIN (minimum value) function 67
minimum value (MIN) function 67
modifying an object file 18, 19
MSG (issue programmer message) directive 131, 132
MSP (memory space) function 67, 68
multi-programmer environment
  absolute mode 85
  example 84–86
  relative mode 86
MXP (memory expansion) function 68

N

names, sections 75, 76
natural logarithm (LOG) function 65
nested sections 76, 77
NOLIST (stop assembly listing) directive 132

For More Information: www.freescale.com

NOTE (include note) directive 132, 133
numeric constants 46, 47

O

object file
    generating 18
    modifying 18, 19
operand field, source statements 38
operation field, source statements 37, 38
operator precedence 50
operators 48–50
OPT (set assembler options) directive 133–139
OPT options 21
options directives 90
options, command line 14–35
options, OPT 21
ORG (initialize memory space) directive 139, 140
overlay
    example 81–84
    manager 80, 81
overlays 79–84
override macro local label (^) directive 96

P

PAGE (advance/size page) directive 140, 141
PMACRO (purge macro definition) directive 141, 142
POS (position of substring) function 68
POS (raise to a power) function 69
position of substring (POS) function 68
PRCTL (send control string to printer) directive 142
precedence, operator 50
processing, assembler 35, 36
programming rules, checking 23–28
project management, software 75–86
purge macro definition (PMACRO) directive 141, 142

R

RADIX (set constant radix) directive 143
raise to a power (POS) function 69
random value (RND) function 69
reading from argument file 17, 18
redirecting source listing 19
reference external symbol (XREF) directive 154
references 11
REL (relative mode) function 69
relative expressions 45
relative mode (REL) function 69
relative mode, multi-programmer environment 86
relocation and sections 78
return sign (SGN) function 70, 71
reverse bits in field (RVB) function 70
RND (random value) function 69
RVB (reverse bits in field) function 70

S

SC100 assembler 13–44
SCP (compare strings) function 70
searching additional directories 20, 21
SECFLAGS (set ELF section flags) directive 143, 144
SECTION (start section) directive 144–147
sections 75–78
    and define symbols 78
    and macros 78
    and relocation 78
    and symbols 77
    fragmented 76, 77
    names 75, 76
    nested 76, 77
SECTYPE (set ELF section type) directive 147, 148
send control string to printer (PRCTL) directive 142
SET (set symbol to value) directive 148
set assembler options (OPT) directive 133–139
set buffer address (BADDR) directive 102, 103
set constant radix (RADIX) directive 143
set ELF section flags (SECFLAGS) directive 143, 144
set ELF section type (SECTYPE) directive 147, 148
set global symbol to value (GSET) directive 125

For More Information: www.freescale.com

set high memory bounds (HIMEM) directive 125, 126
set low memory bounds (LOMEM) directive 129
set symbol size (SIZE) directive 149
set symbol to value (SET) directive 148
set symbol type (TYPE) directive 151
SGN (return sign) function 70, 71
shift and mask (FLD) function 61
SIN (sine) function 71
sine (SIN) function 71
SIZE (set symbol size) directive 149
SNH (hyperbolic sine) function 71
software
    development flow 9–11
    project management 75–86
source listing 40–44
    example 41–43
    final information 43, 44
    redirecting 19
source statements 36–40
    comment field 38
    label field 37
    operand field 38
    operation field 37, 38
    strings 40
    symbol labels 39, 40
    symbol names 39
    variable length execution sets 38, 39
specify macro library (MACLIB) directive 129, 130
specifying endian mode 22, 23
specifying target architecture 21, 22
SQT (square root) function 72
square root (SQT) function 72
start buffer (BUFFER) directive 106, 107
start comment (;) directive 92
start comment lines (COMMENT) directive 107, 108
start conditional assembly (ELSE) directive 118
start conditional assembly (IF) directive 126, 127
start function (@) directive 97, 98
start section (SECTION) directive 144–147
start unreported comment (;;) directive 92
starting the assembler 13, 14
STITLE (initialize program subtitle) directive 149, 150
stop assembly listing (NOLIST) directive 132
storage allocation directives 90
string constants 47
string functions 52
string length (LEN) function 64
strings, source statements 40
substitute location counter (*) directive 98
substitute macro hex value (%) directive 95
substitute macro value (?) directive 94, 95
substitution strings, defining 21
symbol definition directives 89
symbol labels, source statements 39, 40
symbol names, source statements 39
symbols and sections 77

T

TAN (tangent) function 72
tangent (TAN) function 72
target architecture, specifying 21, 22
terms, data analysis 33
TITLE (initialize program title) directive 150
TNH (hyperbolic tangent) function 72, 73
TYPE (set symbol type) directive 151

U

UNDEF (undefine define symbol) directive 151, 152
undefine define symbol (UNDEF) directive 151, 152
UNF (convert fractional to floating point) function 73
use immediate addressing (#) directive 100

W

WARN (issue programmer warning) directive 152, 153
warnings, assembler 166–169
warnings, command-line 163, 164

X

XDEF (define external symbol) directive 153

For More Information: www.freescale.com

XPN (exponential function) function 73
XREF (reference external symbol) directive 154

For More Information: www.freescale.com

For More Information: www.freescale.com